НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.457

«До захисту допущено»
Завідувач кафедри
_____ І.А. Дичка
«___»_____2018 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Програмна модель відмовостійкої обчислювальної системи, що керується потоком даних»

Виконав:
студент VI курсу, групи КП-71мп
Вінник Денис Андрійович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н.,
Жабіна В.В. _____

Рецензент:
Доцент кафедри ОТ факультета ФІОТ, к.т.н., доцент,
Верба О.А. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський) за освітньо-науковою програмою

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення» («Програмне забезпечення комп'ютерних та інформаційно-пошукових систем»)

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ І.А. Дичка

«___»_____2016 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Віннику Денису Андрійовичу

1. Тема дисертації «Програмна модель відмовостійкої обчислювальної системи, що керується потоком даних», науковий керівник дисертації доцент кафедри ПЗКС, к.т.н., Жабіна Валентина Валеріївна затверджені наказом по університету від «___»_____ 2018 р. №_____

2. Термін подання студентом дисертації «14» грудня 2018 р.

3. Об'єкт дослідження: процес обробки даних у відмовостійких обчислювальних системах, що керуються потоком даних.

4. Предмет дослідження: методи забезпечення відмовостійкості в обчислювальних системах, що керуються потоком даних.

5. Перелік завдань, які потрібно розробити:

– аналіз та дослідження існуючих методів і засобів забезпечення відмовостійкості систем з метою виявлення їх недоліків і вибору найбільш доцільного варіанту для СПД;

– аналіз методів динамічної реконфігурації систем;

– розробка архітектури відмовостійкої СПД з використанням асоціативної пам'яті;

– розробка програмного емулятора роботи архітектури;

– експериментальне дослідження ефективності запропонованої моделі за допомогою емулятора.

6. Орієнтовний перелік публікацій:

– Тези доповіді "Програмна модель відмовостійкої обчислювальної системи, що керується потоком даних"

7. Дата видачі завдання «15» жовтня 2016 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.12.2016	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.03.2017	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	16.05.2017	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	14.10.2017	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	15.12.2017	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації; підготовка матеріалів доповіді на конференції ПМК-2018.	20.02.2018	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу; робота над розділом про бізнес-модель	16.04.2018	
8.	Оформлення текстової і графічної частини магістерської дисертації	06.05.2018	

Студент Д.А. Вінник

Науковий керівник дисертації В.В. Жабіна

РЕФЕРАТ

Актуальність. Ефективним підходом до проблеми підвищення надійності систем та достовірності результатів обробки інформації є забезпечення відмовостійкості систем. Найбільш економічним з точки зору апаратури є програмні засоби підвищення відмовостійкості. Нажаль, для систем реального часу часові витрати при такому підході є неприйнятними. В даному випадку більш прийнятними є методи динамічної реконфігурації систем при відмові обладнання. Проблемі забезпечення відмовостійкості систем присвячено багато публікацій, в тому числі, монографій. Тим не менше, ця проблема недостатньо досліджена для СПД через специфіку організації в них обчислювальних процесів.

Таким чином, розробка моделей забезпечення відмовостійкості таких систем є актуальною задачею.

Об'єктом дослідження є процес обробки даних у відмовостійких обчислювальних системах, що керуються потоком даних.

Предметом дослідження є методи забезпечення відмовостійкості в обчислювальних системах, що керуються потоком даних.

Метою дослідження є реалізація програмної моделі для тестування відмовостійкої системи для визначення її оптимальних параметрів.

Методи дослідження. В роботі використовуються елементи теорії графів та алгоритмів, теорії надійності. Використовуються результати експериментів, порівняння характеристик систем.

Наукова новизна роботи полягає в наступному:

1. Запропонована модифікація методу забезпечення відмовостійкості потокових обчислювальних систем, яка відрізняється від існуючих методів врахуванням реальної швидкості виконання кожної операції, що дозволяє зменшити час на виявлення несправного обчислювального модуля. В системах-аналогах враховується лише максимальний час виконання операції.

2. Реалізована гнучка програмна модель розробленої архітектури обчислювальної системи із засобами відмовостійкості.
3. Запропонована система команд для відмово стійкої системи, яка може розширюватися за допомогою спеціального конфігураційного файлу.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований метод за ефективністю не поступається існуючим, а в більшості випадків навіть перевершує. Розроблена програмна модель допоможе спростити розробку реальних проектів.

Апробація роботи. Основні положення і результати роботи доповідалися та обговорювалися на X науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'яти розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, наведено відомості про апробацію результатів.

У першому розділі розглянуто і проаналізовано існуючі рішення, описані їх переваги та недоліки.

У другому розділі запропоновано модифікацію методу забезпечення відмовостійкості за допомогою таймерів, розроблено архітектуру обчислювальної системи, що керується потоком даних.

У третьому розділі описано програмну модель, що реалізує розроблену архітектуру та псевдокод, котрий він оброблятиме.

У четвертому розділі описано результати експериментів та порівнянь між розробленим та існуючими алгоритмами, описано надійність системи.

У п'ятому розділі наведена побудова бізнес-моделі, що обґрунтовує доцільність реалізованої програмної моделі та прогнозує її потенційну прибутковість у майбутньому.

У висновках проаналізовано отримані результати роботи.

У додатках наведена копія презентації та вихідний код.

Робота виконана на 100 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 20 найменувань. У роботі наведено 34 рисунка та 7 таблиць.

Ключові слова: обчислювальні системи, потоки даних, відмовостійкість, асоціативна пам’ять, надійність.

ABSTRACT

Topicality. An effective approach to improving the reliability of systems and the reliability of the results of information processing is to provide fail-safe systems. The most economical in terms of hardware is the software to improve fault tolerance. Unfortunately, for real-time systems, time costs in this approach are unacceptable. In this case, methods for dynamically reconfiguring systems during equipment failure are more acceptable. The problem of providing fail-safe systems is devoted to many publications, including monographs. Nevertheless, this problem is not sufficiently investigated for SPD because of the specifics of the organization in them of computing processes.

Thus, the development of models for providing fault tolerance of such systems is an urgent task.

The object of research is the process of data processing in fault-tolerant computing systems, driven by the flow of data.

The subject of the study is methods for ensuring fault tolerance in data-driven computing systems.

The aim of the study is an implementation of a software model of fault-tolerant computing system driven by a data flow to improve performance and reliability.

Research methods. The paper uses the elements of the theory of graphs and algorithms, the theory of reliability. Used results of experiments, comparison of system characteristics.

The scientific novelty of the work is as follows:

1. The modification of the method for providing failover of flow computing systems is proposed, which differs from the existing methods taking into account the real speed of each operation, which reduces the time to detect a faulty computing module. In analog systems, only the maximum execution time of an operation is taken into account.

2. Flexible software model of the developed architecture of the computer system with means of fault-tolerance is realized.
3. The proposed system of commands for a fail-safe system, which can be expanded with the help of a special configuration file.

The practical value of the results obtained in the work is that the proposed method for efficiency is not inferior to the existing, and in most cases even exceeds. The developed software model will help to simplify the development of real projects.

Test work. The main provisions and results of work were reported and discussed at the Xth International Conference of Masters and Postgraduate Students "Applied Mathematics and Computer", PMK-2018

Structure and scope of work. The master's dissertation consists of an introduction, five sections, conclusions and appendices.

The introduction provides a general description of the work, an assessment of the current state of the problem is made, the relevance of the research direction is substantiated, information about testing the results is given.

The first chapter examines and analyzes existing solutions, describes their advantages and disadvantages.

In the second section, the modification of the method of providing fault tolerance with the help of timers is proposed, and the architecture of the computer system driven by the data flow is developed.

The third section describes the software model that implements the developed architecture and pseudocode, which it will process.

The fourth section describes the results of experiments and comparisons between developed and existing algorithms, describes the reliability of the system.

The fifth section presents the construction of a business model that justifies the feasibility of the implemented software model and predicts its potential profitability in the future.

The conclusions are analyzed the results of work.

The attachments include a copy of the presentation and the source code.

The work is done on 100 sheets, contains 3 attachments and a link to the list of used literary sources of 20 titles. There are 34 drawings and 7 tables in the work.

Keywords: computing systems, data flows, fault-tolerance, associative memory, reliability.

РЕФЕРАТ

Актуальность. Эффективным подходом к проблеме повышения надежности систем и достоверности результатов обработки информации является обеспечение отказоустойчивости систем. Наиболее экономичным с точки зрения аппаратуры являются программные средства повышения отказоустойчивости. К сожалению, для систем реального времени временные затраты при таком подходе являются неприемлемыми. В данном случае более приемлемыми являются методы динамической реконфигурации систем при отказе оборудования. Проблеме обеспечения отказоустойчивости систем посвящено много публикаций, в том числе, монографий. Тем не менее, эта проблема недостаточно исследована для СПД в силу специфики организации в них вычислительных процессов.

Таким образом, разработка моделей обеспечения отказоустойчивости таких систем является актуальной задачей.

Объектом исследования является процесс обработки данных в отказоустойчивых вычислительных системах, управляемых потоком данных.

Предметом исследования являются методы обеспечения отказоустойчивости в вычислительных системах.

Целью исследования является реализация программной модели отказоустойчивой вычислительной системы, управляемые потоком данных для улучшения быстродействия и надежности.

Методы исследования. В работе используются элементы теории графов и алгоритмов, теории надежности. Используются результаты экспериментов, сравнение характеристик систем.

Научная новизна работы заключается в следующем:

1. Предложенная модификация метода обеспечения отказоустойчивости потоковых вычислительных систем, которая отличается

от существующих методов учетом реальной скорости выполнения каждой операции, что позволяет уменьшить время на выявление неисправного вычислительного модуля. В системах-аналогах учитывается только максимальное время выполнения операции.

2. Реализована гибкая программная модель разработанной архитектуры вычислительной системы со средствами отказоустойчивости.

3. Предложенная система команд для отказоустойчивость устойчивой системы, которая может расширяться с помощью специального конфигурационного файла.

Практичесеая ценность полученных в работе результатов заключается в том, что предложенный метод по эффективности не уступает существующим, а в большинстве случаев даже превосходит. Разработана программная модель поможет упростить разработку реальных проектов.

Апробация работы. Основные положения и результаты работы докладывались и обсуждались на X научной конференции магистрантов и аспирантов «Прикладная математика и компьютинг» ПМК-2018

Структура и объем работы. Магистерская диссертация состоит из введения, пяти глав, заключения и приложений.

Во введении дано общая характеристика работы, выполнена оценка современного состояния проблемы, обоснована актуальность направления исследований, приведены сведения об апробации результатов.

В первом разделе рассмотрены и проанализированы существующие решения, описаны их преимущества и недостатки.

Во втором разделе предложено модификацию метода обеспечения отказоустойчивости с помощью таймеров, разработана архитектуру вычислительной системы, управляемой потоком данных.

В третьем разделе описано программную модель, реализует разработанную архитектуру и псевдокод, который он будет обрабатывать.

В четвертом разделе описаны результаты экспериментов и сравнений между разработанным и существующими методами, описано надежность системы.

В пятом разделе приведена построение бизнес-модели, обосновывает целесообразность реализованной программной модели и прогнозирует ее потенциальную прибыльность в будущем.

В выводах проанализированы полученные результаты работы.

В приложениях приведена копия презентации и исходный код.

Работа выполнена на 100 листах, содержит 3 приложения и ссылки на список использованных литературных источников из 20 наименований. В работе приведены 34 рисунка и 7 таблиц.

Ключевые слова: вычислительные системы, потоки данных, отказоустойчивость, ассоциативная память, надежность.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

СПД – системи, що керуються потоком даних. Це системи, що використовують механізм управління обчисленнями, при якому команди виконуються тоді, коли стають доступними їх операнди.

ГПД – граф потоку даних. Це один з найнагляднішихварiантів представлення програми у СПД. Його вузли - це обчислення, які відправляють та приймають повідомлення даних.

Відмовостійкість – це властивість архітектури ОС, що забезпечує виконання заданих функцій у випадках, коли в апаратних і програмних засобах системи виникають відмови.

ОМ – обчислювальні модулі.

СФК – середовище формування команд.

АП – асоціативна пам’ять.

БУ – блок управління.

БК – буферна пам’ять команд.

РТ – тимчасовий регістр для зберігання інформації.

КС – комутаційні середовища.

Динамічна реконфігурація – здатність системи, змінювати апаратні ресурси сервера без необхідності його закриття.

ПДД – пам’ять з довільним адресним доступом.

IDEA – International Data Encryption Algorithm. Це симетричний блочний алгоритм шифрування даних.

ОС – обчислювальна система. Це система з одного або кількох комп’ютерів та відповідним програмним забезпеченням із загальним сховищем даних.

Надійність - властивість об'єкта зберігати в часі у встановлених межах значення всіх параметрів, що характеризують здатність виконувати необхідні

функції в заданих режимах і умовах застосування, технічного обслуговування, ремонту, зберігання і транспортування.

Mono – багато-платформове вільне відкрите втілення системи .NET, яке відповідає стандартам ECMA, включаючи серед іншого і компілятор C#, і Common Language Runtime.

WinForms – Windows Forms. Графічна бібліотека, що є складовою Microsoft .Net, для написання настільних додатків.

ВСТУП

Для систем реального часу, що працюють в контурі автоматичного управління процесами, найважливішими характеристиками є швидкодія та надійність. При реалізації дрібнозернистих алгоритмів ефективними є системи, керовані потоком даних (СПД), що забезпечують автоматичне динамічне розпаралелювання операцій та реалізують прихований паралелізм задач [1]

СПД – це системи, що використовують механізм управління обчисленнями, при якому команди виконуються тоді, коли стають доступними їх операнди. У потокових архітектурах для опису обчислень використовується орієнтований граф потоку даних (ГПД) або dataflow graph. Цей граф складається з вершин (вузлів), що відображають операції, і дуг, що показують потоки даних між тими вершинами графа, які вони з'єднують[2].

Ефективним підходом до проблеми підвищення надійності систем та достовірності результатів обробки інформації є забезпечення відмовостійкості систем. Найбільш економічним з точки зору апаратури є програмні засоби підвищення відмовостійкості. Нажаль, для систем реального часу часові витрати при такому підході є неприйнятними. В даному випадку більш прийнятними є методи динамічної реконфігурації систем при відмові обладнання. Проблемі забезпечення відмовостійкості систем присвячено багато публікацій, в тому числі, монографій [3]. Тим не менше, ця проблема недостатньо досліджена для СПД через специфіку організації в них обчислювальних процесів.

Таким чином, розробка моделей забезпечення відмовостійкості таких систем є актуальною задачею.

В роботі використовуються елементи теорії графів та алгоритмів, теорії надійності. Використовуються результати експериментів, порівняння характеристик систем.

Наукова новизна роботи полягає в наступному:

1. Запропонована модифікація методу забезпечення відмовостійкості потокових обчислювальних систем, яка відрізняється від існуючих методів врахуванням реальної швидкості виконання кожної операції, що дозволяє зменшити час на виявлення несправного обчислювального модуля. В системах-аналогах враховується лише максимальний час виконання операції.
2. Реалізована гнучка програмна модель розробленої архітектури обчислювальної системи із засобами відмовостійкості.
3. Запропонована система команд для відмово стійкої системи, яка може розширюватися за допомогою спеціального конфігураційного файлу.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований метод за ефективністю не поступається існуючим, а в більшості випадків навіть перевершує. Розроблена програмна модель допоможе спростити розробку реальних проектів.

1. МЕТОДИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РЕАЛІЗАЦІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ У РОЗПОДІЛЕНИХ СИСТЕМАХ

1.1. Загальні теоретичні відомості

1.1.1 Паралельні обчислення

Паралельні обчислення - це тип обчислення, в якому багато обчислень або виконання процесів виконуються одночасно. Великі проблеми часто можна розділити на менші, що потім можна вирішити одночасно. Існує кілька різних форм паралельних обчислень: бітовий рівень, рівень інструкцій, даних та паралельність завдань. Паралельність вже давно працює на високопродуктивних обчисленнях, але воно набуває Паралельні обчислення — це форма обчислень, в яких кілька дій проводяться одночасно. Ґрунтуються на тому, що великі задачі можна розділити на кілька менших, кожну з яких можна розв'язати незалежно від інших[4].

Є кілька різних рівнів паралельних обчислень: бітовий, інструкцій, даних та паралелізм задач. Паралельні обчислення застосовуються вже протягом багатьох років, в основному в високопродуктивних обчисленнях, але зацікавлення ним зросло тільки недавно, через фізичні обмеження зростання частоти. Оскільки споживана потужність (і відповідно виділення тепла) комп'ютерами стало проблемою в останні роки, паралельне програмування стає домінуючою парадигмою в комп'ютерній архітектурі, основному в формі багатоядерних процесорів[5].

Паралельні комп'ютери можуть бути грубо класифіковані згідно з рівнем, на якому апаратне забезпечення підтримує паралелізм: багатоядерність, багатопроцесорність — комп'ютери, що мають багато обчислювальних елементів в межах одної машини, а також кластери, MPP, та ґрід — системи що використовують багато комп'ютерів для роботи над одним завданням.

Спеціалізовані паралельні архітектури іноді використовуються поряд з традиційними процесорами, для прискорення особливих задач[6].

Програми для паралельних комп'ютерів писати значно складніше, ніж для послідовних, бо паралелізм додає кілька нових класів потенційних помилок, серед яких найпоширенішою є стан гонитви. Комунікація, та синхронізація процесів зазвичай одна з найбільших перешкод для досягнення хорошої продуктивності паралельних програм.

Максимальний можливий приріст продуктивності паралельної програми визначається законом Амдала (рис. 1.1)[7].

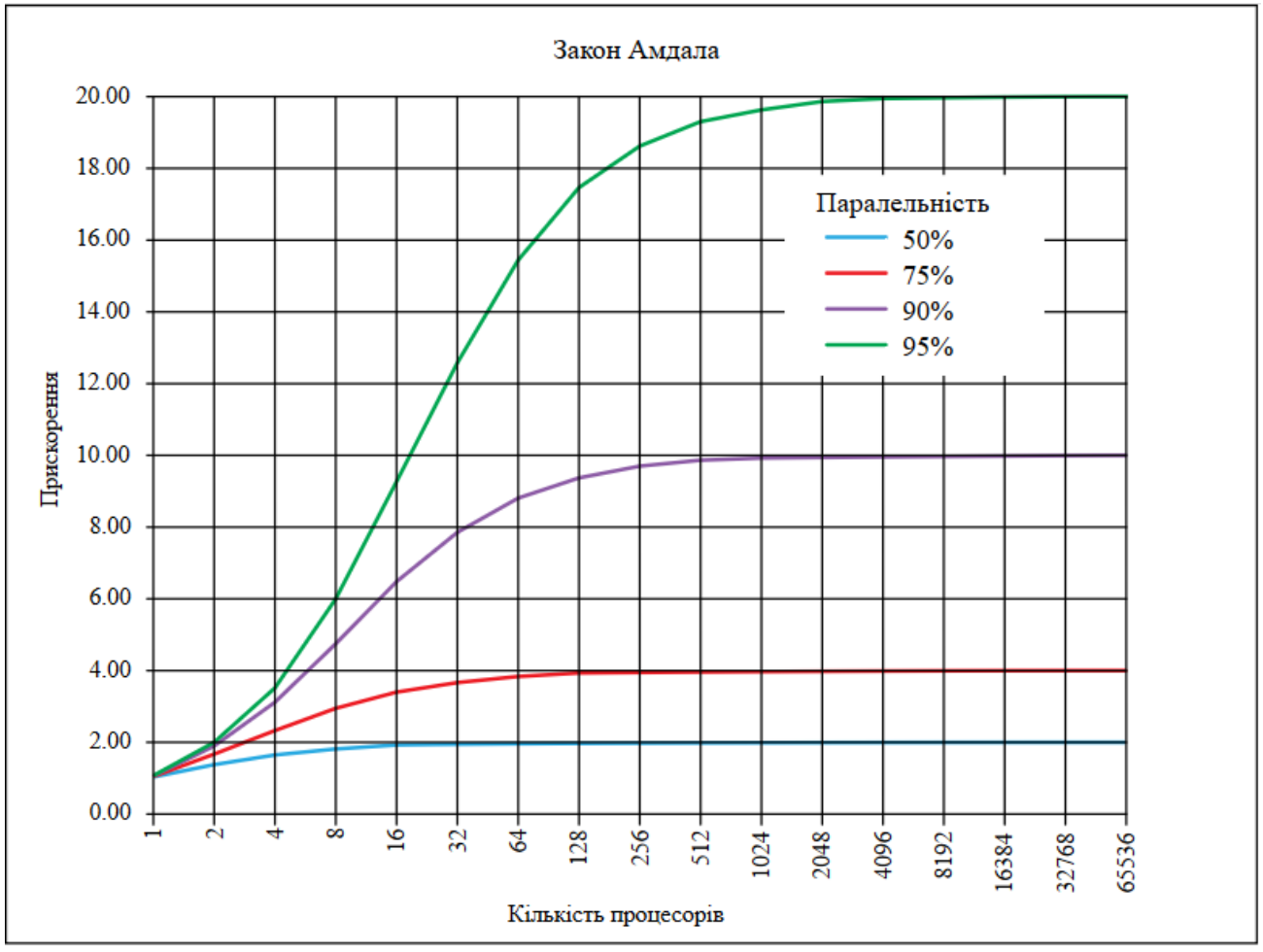


Рисунок 1.1 – Графічне зображення закону Амдала.

1.1.2 Системи, що керуються потоком даних

Програмування для систем, що керуються потоками даних вводить нову парадигму програмування, яка внутрішньо представляє програми як направлений граф, аналогічно діаграми потоку даних. Програми представлені у вигляді набору вузлів (так званих блоків) з вхідним та/або вихідними портами в них. Ці вузли можуть бути джерелами, потоками або блоками обробки інформації, що протікає в системі. Вузли поєднані направленими дугами, які визначають потік інформації між ними. Більшість візуальних мов програмування, які використовують архітектуру на основі блоків для представлення їх робочого пототку насправді спираються на DFP[8].

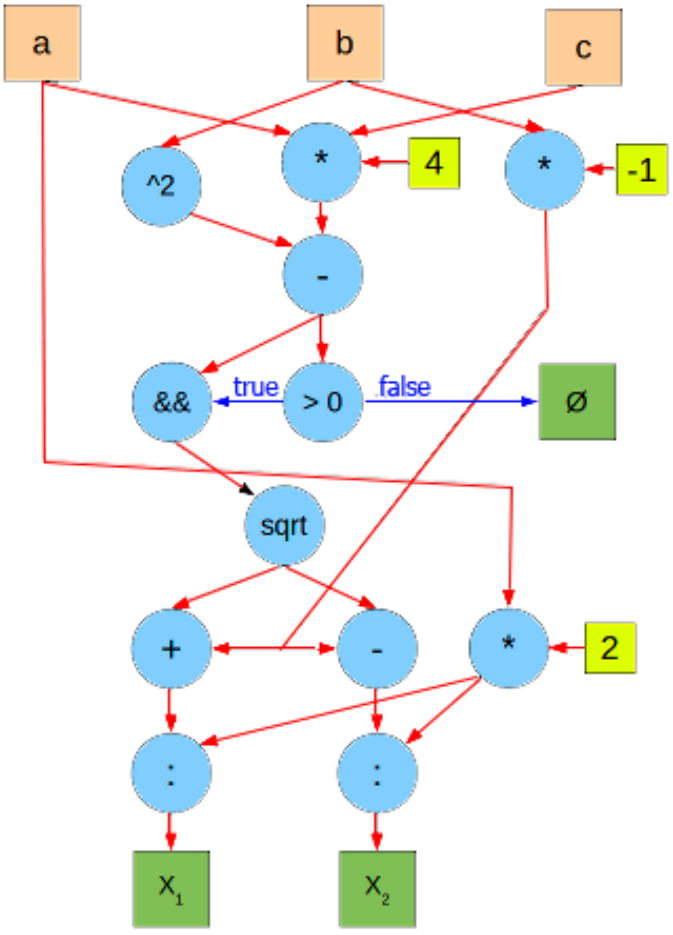


Рисунок 1.2 – Приклад графу потоку даних.

Першою перевагою є наявність мов візуального програмування, що полегшує роботу програмістів з інструментом, який завдяки своєму спрощеному інтерфейсу може забезпечити швидке прототипування та впровадження певних систем. Мови візуального програмування, як відомо, спрощують процес впровадження програмування кінцевого користувача, де користувач програми може певним чином змінити поведінку програми. Існує багато мов, що надають такі можливості. Візуальне програмування було успішно прийнято як досвідченими програмістами, так і нетехнічними користувачами комп'ютера, які можуть використовувати цю мову як інструмент для розширення існуючої програми або написання з нуля[9].

Другим пунктом на користь DFP є неявне досягнення паралельності.

У внутрішньому представленні програми кожний вузол є незалежним обчислювальним блоком, який не створює жодних сторонніх ефектів, тобто працює незалежно від інших. Така модель виконання дозволяє вузлам виконуватись, як тільки до них надходять дані, без можливості створення тупикових точок, так як відсутня залежність даних у всій системі. Це основна особливість моделі потоку даних, що виключає необхідність того, щоб програмісти поводились з проблемами паралельності, такими як семафори, або ручне виведення та управління потоками. Така особливість значно підвищує продуктивність програми, коли вона виконується на багатоядерному процесорі, загальноприйнятій архітектурі в даний час, не впроваджуючи жодної додаткової роботи для програміста[10].

Ці два основні моменти з DFP дозволяють вважати, що ця парадигма повинна бути частиною знань будь-якого розробника, яка дає йому змогу використовувати її в сценаріях, якщо це необхідно.

1.1.3 Відмовостійкість

Відмовостійкість – це властивість архітектури ОС, що забезпечує виконання заданих функцій у випадках, коли в апаратних і програмних засобах системи виникають відмови [11]

Для її забезпечення в системі повинні бути передбачені наступні етапи:

- Контроль стану ОС. Завдання етапу: визначення факту присутності однієї або декількох помилок в системі
- Діагностування. В рамках даного етапу виявляється характер несправності і визначаються несправні ресурси.
- Локалізація несправних ресурсів.
- Реконфігурація. В рамках цього етапу в систему вносяться ряд змін, які дозволяють подальше рішення задачі.
- Відновлення обчислювального процесу.
- Процес діагностування систем може бути реалізований як самодіагностування, тобто об'єкт діагностування сам визначає свій стан і знаходить несправні компоненти.

Самодіагностування може бути засноване на двох принципах:

- принцип апаратно захищених ядер, що розширюються;
- принцип розподіленого ядра.

Одним з основних методів тестування модулів є контроль часових інтервалів (таймерний контроль). У цьому випадку заздалегідь існують чітко визначені проміжки часу, коли один модуль очікує сигналу від іншого. Модуль, що очікує сигнал – контролюючий. Модуль, який повинен послати сигнал – контрольований. Рішення про відмову контрольованого модуля приймається в тому і тільки в тому випадку, якщо очікуваний сигнал не отримано, але є сигнал від таймера про закінчення контрольованого інтервалу[12].

1.2. Існуючі методи підвищення ефективності роботи СПД

1.2.1. Обчислювальна система з асоціативною пам'яттю

До складу системи входять обчислювальні модулі (ОМ), середовище формування команд (СФК) на базі асоціативної пам'яті (АП), регістри для тимчасового зберігання інформації (РВ), блок управління (БУ), буферна пам'ять (БП) і комутаційні середовища (КС).

Формування команд проводиться наступним чином. Актори і дані в будь-якому порядку надходять через КС1 в СФК. З використанням процедури адресного запису актори і дані для і-й команди записуються в одну клітинку АП за адресою Ii, яка збігається для всіх об'єктів однієї команди. Одночасно з записом актора і даних встановлюються признаки їх наявності у відповідних розрядах комірок пам'яті. Значення ознак вказує на наявність в АП готової команди для ОМ. Готова команда зчитується з АП за допомогою процедури асоціативного читання і переписується в БП типу FIFO. Цикл безадресного читання даних з БП значно менше циклу асоціативного читання з АП, що дає можливість прискорити обчислення.

Вільний ОМj приймає команду з БП і приступає до її виконання. Адреса Ii на час виконання команди зберігається в РВ. Одночасно з цим скидаються в нульове стан ознаки в відповідну комірку АП і блокується запис нової інформації в зазначену комірку. Поки при-знаки знову НЕ будуть встановлені в одиницю, команда вважається неготовою і не може бути зчитана повторно іншим ОМ. Разом з цим в БУ запускається таймер Tj на час, який достатній для виконання самої найдовшої команди. Число таймерів має дорівнювати числу ОМ.

Після успішного виконання команди результат з ОМj через КС1 записується в АП за адресою Ni. Разом з цим скидається таймер Tj і

розблокується адресний запис в комірку з адресою Ii, тобто ця комірка може бути використана для формування іншої команди з відповідним ім'ям.

У разі відмови ОМj спрацьовується таймер Tj (закінчується ліміт часу на виконання команди). ОМj відключається від КС і команда активується шляхом установки признаків в комірку АП з адресою Ii, який зберігався в РВj. Вільний ОМ зчитує цю команду і вона виконується по-повторних, як зазначено вище.

Оскільки підготовка алгоритму до реалізації для потокової системи виконується без урахування конкретного числа ОМ в системі, то є можливість продовження обчислення до тих пір, поки в системі буде залишатися хоча б один працездатний ОМ.

Основним недоліком системи є складність асоціативної пам'яті і велика тривалість циклу асоціативного пошуку даних, що обмежує обсяг пам'яті і знижує продуктивність системи[13].

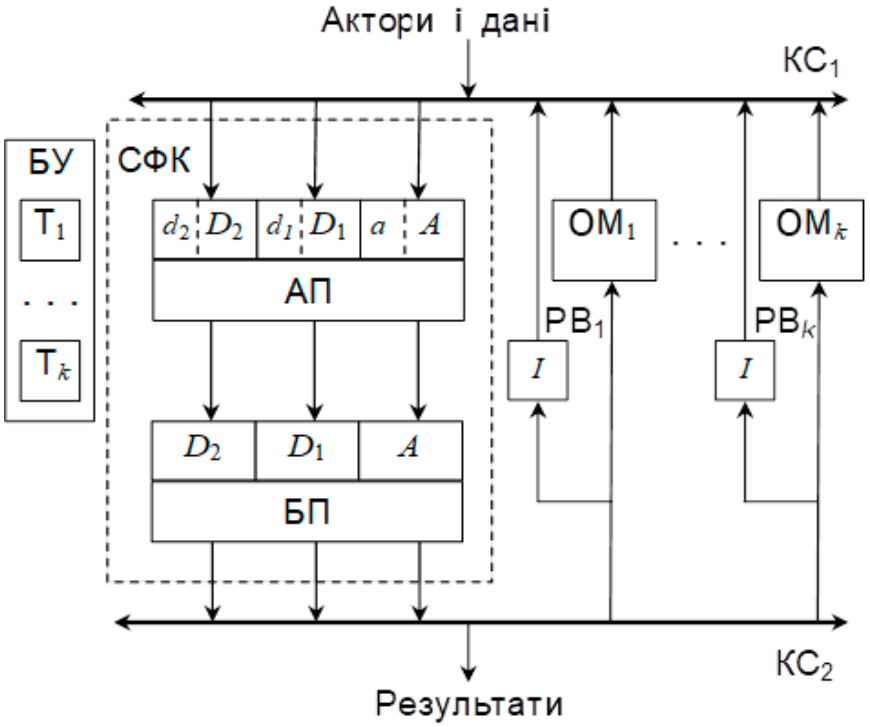


Рисунок 1.3 – Існуюча обчислювальна система з використанням асоціативної пам’яті.

Труднощі реалізації асоціативних запам’ятовуючих пристроїв великого обсягу призводить до необхідності їх емуляції із застосуванням інших технічних засобів, наприклад, спеціалізованих процесорів, що дозволяє збільшити об'єм пам'яті, але істотно знижує продуктивність, оскільки процес емуляції вимагає великих затрат часу. Крім того, при такій організації системи неможливо використовувати осередки АП після зчитування команди для інших команд, що потрібно, наприклад, при ітераційних обчисленнях і обчисленнях в конвеєрному режимі. Все це знижує ефективність паралельних обчислень.

Схема зображена на рисунку 1.3.

1.2.2. Обчислювальна система з пам’яттю з довільним доступом

Команда починає формуватися в комірках ПДД, два розряду яких (d і a) є ознаками, що вказують на наявність в ПДД одного операнда D1 і актора A з часом очікування його результату виконання T. Фактично T і A можуть розглядатися як один інформаційний об'єкт. Запис зазначених об'єктів для і-ї команди здійснюється за адресою Ii. При надходженні з КС1 другого операнда D2 для даної команди, він записується безпосередньо у відповідні розряди регістра буферної пам'яті (БП) типу FIFO, куди одночасно з ПДД переписуються A, T і D1. При цьому комірка ПДД звільняється(чому так можна, буде описано нижче).

Таким чином, в БП формується готова для виконання команда, яка потім через КС2 передається у вільний ОМ. При цьому вміст відповідної комірки ПДД зберігається для забезпечення повторного формування команди в разі відмови ОМ. Одночасно з надходженням команди в ОМj вона повністю записується в регістр тимчасового зберігання РВj на вхідній шині відповідного обчислювача. Саме через те, що записується уся команда у відповідний РВ, то немає необхідності залишати зайву інформацію в ПДД. Уся необхідна інформація для повторної операції, в разі відмови процесора, буде присутня в РВ. При успішному

виконанні операції, тобто коли час її виконання не перевищило ліміт часу виконання команди в ОМj встановлений відповідним таймером Tj, що знаходиться у відповідному тимчасовому регістрі РВj і запускається відразу із записом в нього та ОМj команди. Тож немає необхідності тримати зайві таймер в БУ. Результат операції надходить через КС1 в ПДД за адресою Ni.

Якщо при виконанні операції виникла відмова ОМj, тобто ліміт часу виконання операції, заданий відповідним Tj, був перевищений, ОМj вважається несправним і блокується (відключається від КС1 і КС2), а значення D2 з РВj знову передається в СФК. Оскільки значення A і D1 загублені не були (відповідна комірка ПДД не була змінена), то команда повторно записується в БП, що дає можливість реалізувати наступну успішну спробу виконання команди в справному ОМ.

Розглянутий підхід дозволяє скоротити час реконфігурації і відновлення системи при відмові ОМ, що є дуже важливим фактором для систем реального часу[14].

Схема зображена на рисунку 1.4.

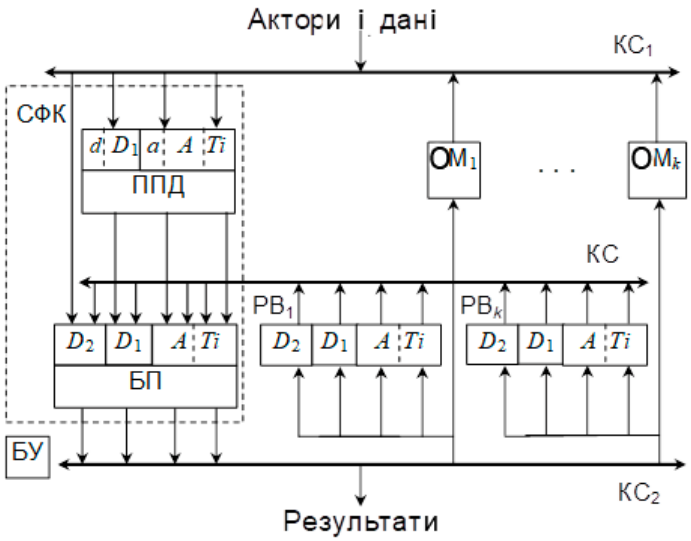


Рисунок 1.4 – Існуюча обчислювальна система з пам’яттю з довільним доступом.

1.3. Сучасні симулятори

1.3.1. Xilinx ISE

Xilinx ISE (Інтегроване середовище синтезу) - це програмний інструмент, розроблений компанією Xilinx для синтезу та аналізу конструкцій HDL, що дозволяє розробнику синтезувати ("компілювати") свої проекти, виконувати аналіз часу, аналізувати RTL діаграми, моделювати реакцію дизайну на різні події та налаштування цільового пристрою програмістом.

Xilinx ISE є дизайнерським середовищем для продуктів FPGA від Xilinx і тісно пов'язаний з архітектурою таких чіпів і не може використовуватися з продуктами FPGA від інших постачальників. Xilinx ISE в основному використовується для синтезу та конструювання схем, тоді як ISIM або логічний симулятор ModelSim використовується для тестування на системному рівні. Інші компоненти, що постачаються разом з Xilinx ISE, включають в себе пакет Embedded Development Kit (EDK), комплект розробника програмного забезпечення (SDK) та ChipScope Pro.

Основним користувальницьким інтерфейсом ISE є Project Navigator, який включає в себе ієрархію дизайну (джерела), редактор вихідного коду (Workplace), вихідну консоль (Transcript) та дерево процесів (Processes).

Ієрархія дизайну складається з дизайнерських файлів (модулів), інтерпретація яких залежить від ISE і відображається у вигляді деревоподібної структури. Для одночіпних конструкцій може бути один основний модуль, з іншими модулями, включеними основним модулем, подібним до основної підпрограми в програмах C++. Дизайн обмеження вказуються в модулях, які включають контактну конфігурацію та відображення.

Ієрархія процесів описує операції, які ISE буде виконувати в поточному активному модулі. Ієрархія включає в себе функції компіляції, їх функції залежностей та інші утиліти. Вікно також позначає проблеми або помилки, що виникають з кожною функцією[15].

Вікно Transcript надає статус поточно запущених операцій та повідомляє інженерів про проблеми дизайну. Такі проблеми можуть бути відфільтровані, щоб відображати Попередження, Помилки або і те, і інше.

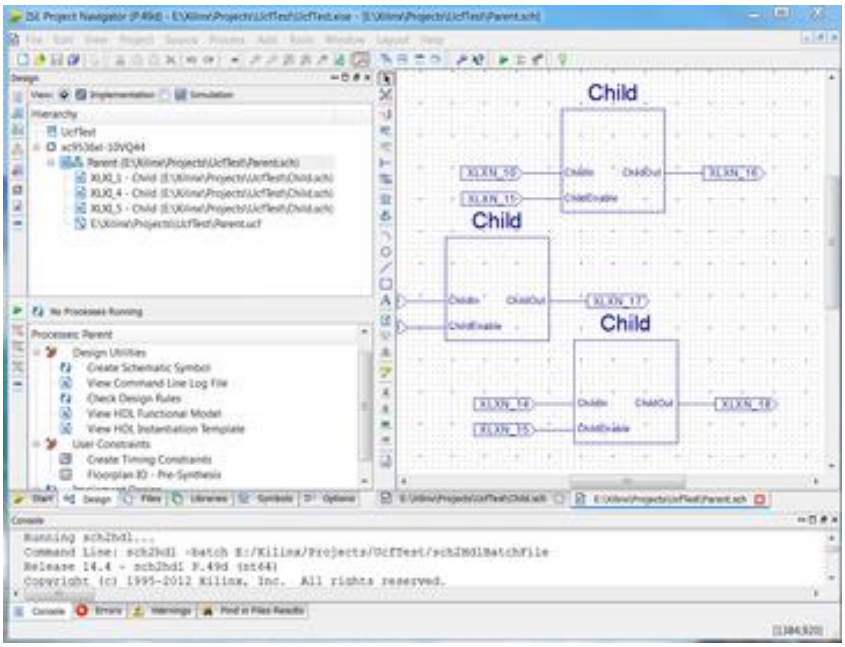


Рисунок 1.5 – Інтерфейс Xilinx ISE.

Тестування на рівні системи може виконуватися за допомогою ISIM або логічного симулятора ModelSim, і такі тестові програми також повинні бути написані на мовах HDL. Програми випробувальних стендів можуть включати симульовані форми сигналу вхідного сигналу або монітори, які спостерігають та перевіряють виходи випробуваного пристрою.

ModelSim або ISIM можуть використовуватися для виконання таких типів моделювання:

- Логічна перевірка, щоб забезпечити, що модуль виробляє очікувані результати
- Поведінкова перевірка, щоб перевірити логічні та часові питання
- Симуляція пост-місця та маршруту, щоб перевірити поведінку після розміщення модуля в межах реконфігурованої логіки FPGA.

1.3.2. Icarus Verilog

Icarus Verilog робить практичну роботу з використання Verilog; він збирає весь письмовий вихідний код дизайну Verilog, перевіряє на наявність помилок кодування та записує зібрані файли дизайну. Він допомагає отримати доступ до вихідних файлів, зібраних у бібліотеках, об'єднати разом модулі по всій вихідній копії та записувати складені результати.

Icarus Verilog призначений для роботи в основному як симулятор, хоча можливості його синтезу поліпшуються. Компілятор Icarus Verilog зазвичай компілює програму Verilog у виконуваний файл, який можна запустити для виконання фактичного моделювання. Вона також включає в себе програмне забезпечення, яке керує скомпільованим моделюванням та генерує виходи тексту або сигналу, запитані програмістом.

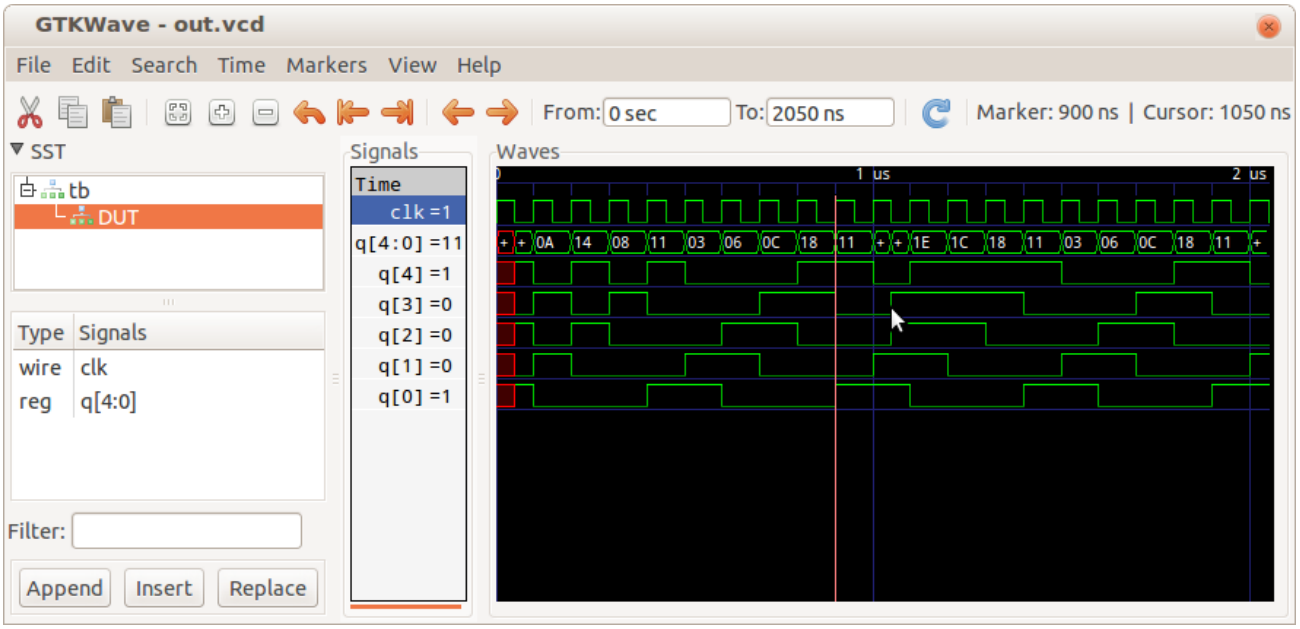


Рисунок 1.6 – Інтерфейс GTKWave.

Окрім формату симуляції за замовчуванням, Icarus Verilog може генерувати різні зовнішні формати. Під час створення netlists, компілятор повинен знати синтаксис файлу нетліст і примітиви цільової технології або пристрою. Цільові формати повинні бути більш конкретними, ніж просто формат

списку. Наприклад, недостатньо запитати EDIF netlist; генератор коду повинен знати, які примітиви можуть бути включені в netlist. Тому Icarus Verilog включає в себе розширюваний інтерфейс генератора коду, який може бути використаний для приєднання нових форматів виводу до компілятора[16].

Результуючий файл можна переглянути за допомогою програми GTKWave. Дана програма призначена для перегляду часових діаграм сигналів, отриманих в результаті симуляції проектів. Інтерфейс показано на рисунку 1.6.

1.4. Висновки

В даному розділі було надано загальні теоретичні відомості про паралельні обчислення, системи, що керуються потоками даних, поняття відмовостійкості та надійності. Розглянуто та проаналізовано існуючі методи підвищення ефективності роботи систем, що керуються потоками даних. У якості прикладів було описано дві системи: одна, що використовує асоціативну пам’ять, інша використовує пам’ять з довільним адресним доступом. У результаті аналізу були визначені можливі варіанти покращення їх можливостей. А саме, визначено, що більш ефективним, для даної мети, рішенням буде використання асоціативної пам’яті. Порівняно з існуючою системою з асоціативною пам’яттю, нова буде розширювати поняття актора. Він також міститиме час свого виконання для використання в таймерах. У існуючому методі завжди використовувався час виконання найважчої операції. Дане рішення дозволить пришвидшити процес діагностування несправного обчислювального модуля.

Також розглянуто сучасні симулятори: комерційний Xilinx ISE та Icarus Verilog, що має відкритий вихідний код. ISE має дуже широку функціональність, котра не є необхідною для більшості задач. Така громіздкість є суттєвим недоліком при проектуванні невеликих систем. Також даний продукт є комерційним, що теж може стати деяким недоліком. Icarus Verilog не має інтерфейсу користувача, що дещо ускладнює роботу з ним. У результаті роботи буде згенерований файл, котрий потім можна буде переглянути за допомогою

вільного програмного забезпечення GTKWave, що прийматиме на вхід результуючий файл з Icarus. Дана програма має графічний інтерфейс, тож можна зручно переглянути необхідні результати. Розроблений та описаний у наступних розділах емулятор буде некомерційним, досить легким та зі спрощеною функціональністю. На відміну від Icarus Verilog матиме свій графічний інтерфейс, тож результат можна буде побачити відразу.

2. АРХІТЕКТУРА ВІДМОВОСТІЙКОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ, ЩО КЕРУЄТЬСЯ ПОТОКОМ ДАНИХ

2.1. Опис архітектури

У склад системи (рис. 2.1) входять обчислювальні модулі (ОМ), середовище формування команд (СФК) на базі асоціативної пам’яті (АП), регістри для тимчасового зберігання інформації (РТ), блок управління (БУ), буферна пам’ять команд (БК) та комутаційні середовища (КС).

В даній роботі розглядається наступний підхід до забезпечення відмовостійкості СПД. Для визначення несправностей ОМ використовується контроль часових інтервалів. Контролюючим об’єктом є апаратно захищене ядро системи, під яким розуміють СФК та БУ.

Відома відмовостійка СПД, в якій також застосовується таймерний контроль ОМ описана в розділі 1.2.1. Її недоліком є велика тривалість інтервалу контролю ОМ і реконфігурації системи. Це пояснюється тим, що таймери для всіх ОМ налаштовуються на час виконання найтривалішої операції, що може перевищувати середній час виконання операцій. Крім того, для повторного виконання команда потрапляє у чергу готових команд останньою, що також збільшує час отримання результату.

Покажемо можливість підвищення ефективності обчислень в СПД з асоціативною пам’яттю у середовищі формування команд. Для визначеності будемо вважати, що система команд містить двохадресні команди.

Формування команд відбувається наступним чином. Актори і дані (токени) в будь-якому порядку поступають через КС1 в СФК. З використанням процедури адресного запису актори і дані для *i*–ї команди записуються в одну комірку АП за адресою I_i, що співпадає для всіх об’єктів одної команди. Одночасно із записом актора і даних встановлюються ознаки їх наявності у відповідних розрядах комірки пам’яті (d_2, d_1, a). Значення ознак $d_2 d_1 a = 111$ вказує на наявність в АП готової команди для ОМ у формі

$$I:\langle D_2;D_1;A.T\rangle,$$

де I – адреса комірки АП, D_1,D_2 – дані; A,T – актор (код операції) з тривалістю виконання команди.

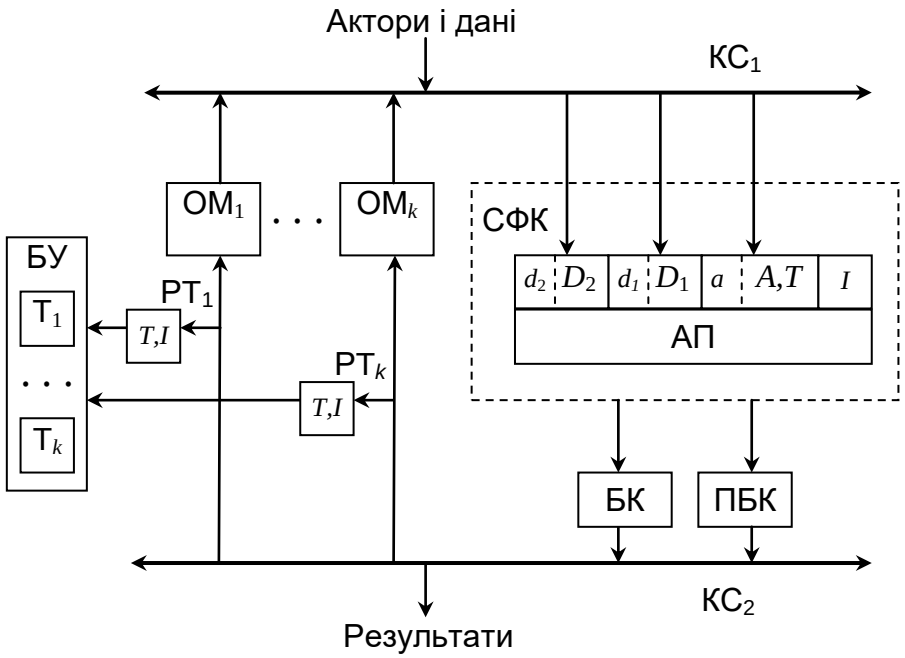


Рисунок 2.1 – Схема архітектури відмовостійкої СПД.

Готова команда зчитується з АП за допомогою процедури асоціативного читання і переписується до БК типу FIFO. Цикл безадресного читання даних з БК є значно меншим циклу асоціативного читання з АП, що дає можливість прискорити обчислення.

Вільний ОМ_j приймає команду з БК. Одночасно в РТі приймається адреса команди та тривалість відповідного інтервалу її виконання. Код ініціалізації інтервалу переписується у відповідний таймер БУ. Число таймерів у БУ має бути рівним кількості ОМ. Адреса I_i на час виконання команди зберігається в РТ_j. Одночасно з цим скидуються у нульовий стан ознаки у відповідній комірці АП і блокується запис нової інформації у вказану комірку. Доки ознаки знов не будуть

встановлені в одиницю, команда вважається неготовою і не може бути виконана повторно іншим ОМ. Разом з цим у БУ запускається таймер T_j і починається виконання команди.

Після успішного виконання команди результат через КС1 записується в АП. Разом з цим встановлюється у нуль таймер T_j та розблоковується адресний запис у комірку з адресою I_i, тобто ця комірка може бути використана для формування іншої команди із відповідним ім'ям (адресою).

У разі відмови $\mathrm{ОМ}_j$ спрацьовує таймер T_j (закінчується ліміт часу на виконання команди). Команда активується шляхом встановлення ознак $d_2 d_1 a = 111$ у комірці АП з адресою I_i, яка зберігалася в RT_j. Ця готова команда записується у пріоритетний буфер команд ПБК. Вільний ОМ зчитує команду з ПБК і вона виконується повторно, як вказано вище. При передачі команд до ОМ спочатку перевіряється наявність команд у ПБК, що запобігає можливості виникнення ситуації глухого кута, коли без результату даної команди неможна продовжити обчислення.

В запропонованому підході до побудови системи кожній команді виділяється необхідний час її виконання, що прискорює реконфігурацію системи при відмові обчислювальних модулів, тобто зменшує час реалізації заданого алгоритму вирішення задачі.

Оскільки підготовка алгоритму для потокової системи виконується без урахування конкретної кількості ОМ в системі, то існує можливість продовження обчислень доти, поки в системі буде залишатися хоча би один працездатний ОМ.

2.2. Робота буферної та асоціативної пам’яті

В даному підрозділі виведемо співвідношення тривалості циклів читання/запису різних типів пам'яті, використанням яких зумовлено описаною вище архітектурою.

На рисунках 2.2 та 2.3 приведені загальні структурні схеми буферного і асоціативного запам'ятовуючих пристроїв.

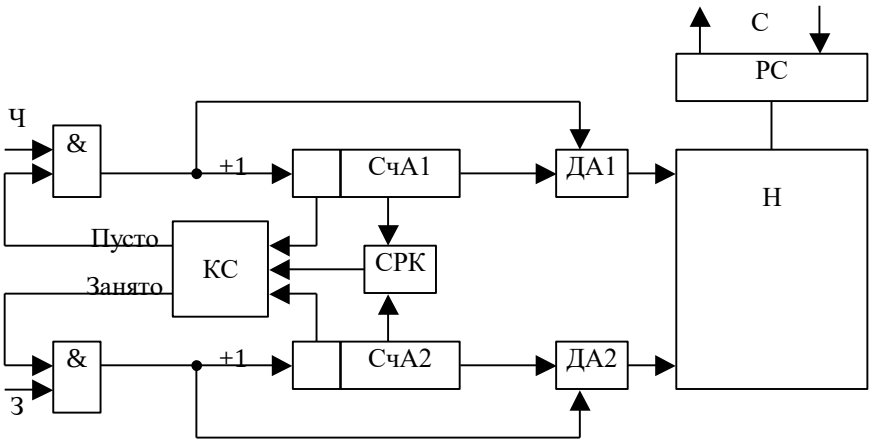


Рисунок 2.2 – Схема буферного запам’ятовуючого пристрою.

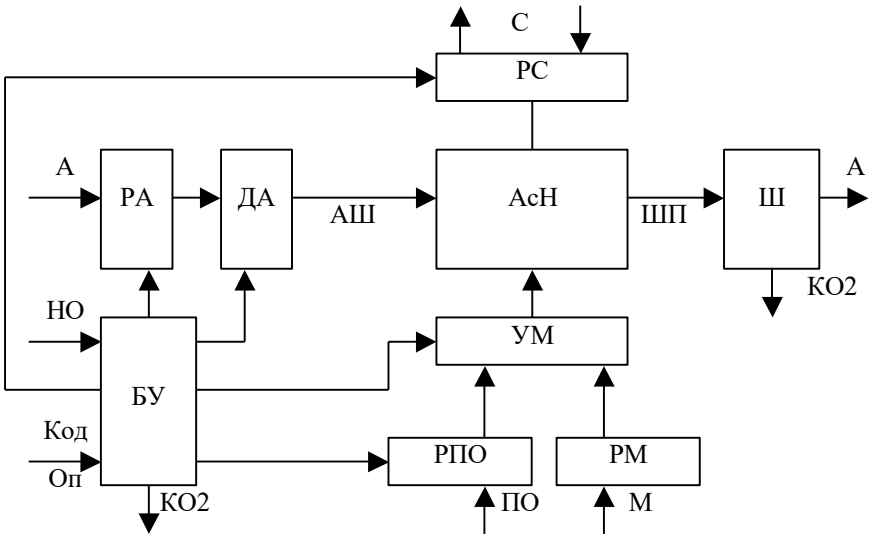


Рисунок 2.3 – Схема асоціативного запам’ятовуючого пристрою.

Як видно з вищенаведених схем, більш складною є схема асоціативного запам’ятовуючого пристрою. І це не дивно, адже на пристрій цього типу лягає крім завдання запису і читання за адресою, також читання і запис за ознакою. Отже, цей пристрій має також здійснювати пошук інформації по її вмісту. Розглянемо внутрішній устрій запам'ятовуючих елементів, які використовуються в накопичувачах буферної пам'яті (рис. 2.4) і в асоціативному накопичувачі (рис. 2.5)[17].

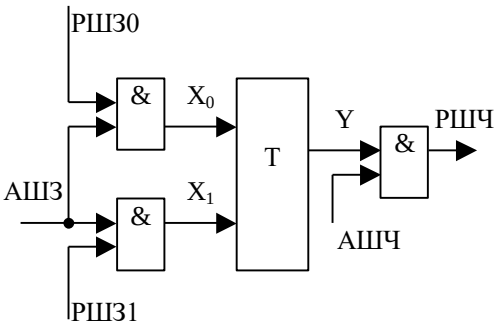


Рисунок 2.4 – Логічна структура звичайного запам’ятовуючого елемента.

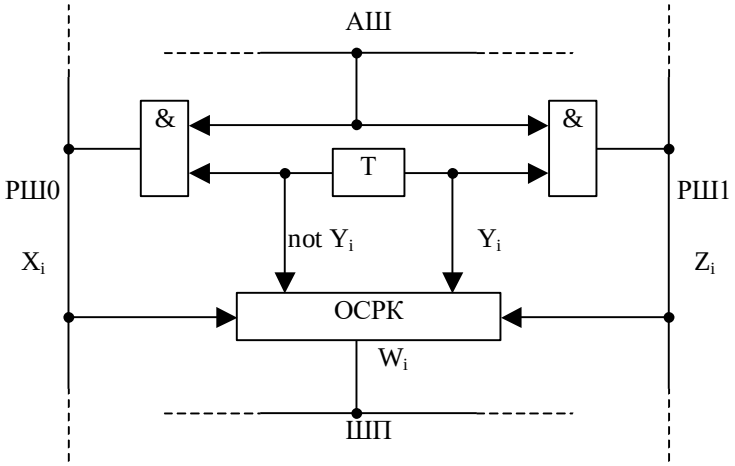


Рисунок 2.5 – Логічна структура асоціативного запам’ятовуючого елемента.

Тут теж можна побачити, що структура асоціативного запам'ятовуючого елемента трохи складніше того, що використовується в буферній пам'яті. Також в асоціативному запам'ятовуючому елементі вихід з однорозрядної схеми рівності кодів (об'єднане субрегіональне командування) повинен бути підключений до шини пошуку (ШП) так, щоб сигнали від окремих запам'ятовуючих елементів, що формують одну комірку, реалізовували кон'юнкцію.

Спираючись на алгоритми роботи всіх вищеописаних типів запам'ятовуючих пристроїв виведемо формули для циклу звернення до пам'яті.

Цикл звернення до буферної пам'яті буде становити:

$$t_{буф} = 3 * t_{триг} + t_{дш} + 3 * t_i = t_{адр} + 2 * t_i.$$

А цикл звернення до асоціативної пам'яті:

$$t_{асоц} = 5 * t_{триг} + t_{дш} + t_ш + 3 * t_i + t_{або} = t_{буф} + 2 * t_{триг} + t_ш + t_{або},$$

де

- $t_{триг}$ - час перемикання тригера,
- $t_{дш}$ - затримка на дешифраторі,
- t_i - затримка на елементі «І»,
- $t_ш$ - затримка на шифраторі,
- $t_{або}$ - затримка на елементі «АБО».

Як видно з отриманих співвідношень час звернення до асоціативної пам'яті значно вище часу звернення до буферного запам'ятовуючого пристрою. Це ще раз підтверджує наші зауваження з приводу того, що при оцінці продуктивності систем, здійсненої в попередньому підрозділі, слід враховувати різну величину швидкодії різних типів пам'яті. Те, що і в розробленій системі була використана асоціативна пам'ять, незначно вплине на продуктивність системи в цілому по причині того, що вона надає свої переваги для інших моментів, коли необхідний швидкий пошук за ознаками.

2.3. Формат слів даних

Основні слова, котрі використовуються у розробленій архітектурі це актори та операнди. Їх звичайні формати приведені на рисунках 2.6 та 2.7[18].

Тип слова	Операція	Операнд	Код операції	Наступна операція	Введення	Порядок введення	Конст

Рисунок 2.6 – Формат актора.

Тип слова	Операція	Операнд	Значення	Введення	Порядок введення	Конст

Рисунок 2.7 – Формат операнда.

Актори складаються з таких полів:

тип слова – 1,

операція – адреса в пам’яті акторів,

операнд – номер операнда, що буде вирахований,

код операції – визначає код операції актора,

наступна операція – адреса наступної операції,

введення – визначає пристрій введення з якого буде очікуватися введення слова,

порядок введення – має виключити ситуацію, коли актор буде зачитаний після операндів,

конст – чи буде результат операції константим.

Операнди складаються з таких полів:

тип слова – 0,

операція – адреса в пам’яті операндів,

операнд – порядковий номер операнда при обрахунку (важливо для віднімання, ділення і тд),

значення – містить конкретне значення опернада,

введення – визначає пристрій введення з якого буде очікуватися введення слова,

порядок введення – має виключити ситуацію, коли операнд буде зачитаний раніше актора,

конст – чи є операнд константим.

2.4. Програмна модель для розробленої архітектури

2.4.1. Загальні відомості

Для тестування розробленої архітектури було розроблено емулятор, що емулює роботу розробленої відмовостійкої системи, що керується потоком даних. Він включає компілятор для розробленої псевдомови, модулі, що емулюють роботу СПД та графічний інтерфейс для конфігурації та поетапного перегляду роботи СПД.

Розроблений псевдокод може бути розширений необхідними командами, котрі можуть бути виконані обчислювальними модулями. Це зроблено для того, щоб конкретний користувач міг описати можливості своєї системи, включаючи набір команд та час їх виконання. Емуляція роботи виконується абстрактно і необхідна лише для оцінки ефективності виконання алгоритму.

Для розробки емулятора обрано Mono, та Winforms для графічного інтерфейсу.

2.4.2. Mono

Mono, платформа розробки з відкритим кодом, заснована на .NET Framework, дозволяє розробникам створювати міжплатформенні програми з

покращеною продуктивністю розробників. Реалізація .NET Mono заснована на стандартах ECMA для C# та загальної інфраструктури мови.

Підтримувані раніше компанії Novell, Xamarin, а тепер і Microsoft та .NET Foundation, проект Mono має активну та захопливу спільноту. Mono включає в себе як інструменти розробника, так і інфраструктуру, необхідну для запуску клієнтських та серверних додатків .NET[19].

Є кілька компонентів, які складають моно:

- Компілятор C # - Компілятор C # для Mono є повною версією для C # 1.0, 2.0, 3.0, 4.0, 5.0 та 6.0 (ECMA). Хороший опис особливостей різних версій доступний у Вікіпедії.
- Mono Runtime - час виконання реалізує інфраструктуру загальної мови ECMA (CLI). Час виконання забезпечує компілятор "Простий у часі" (JIT), компілятор "Ahead-of-Time" (AOT), бібліотечний завантажувач, збирач сміття, систему різання та функціональність сумісності.
- Бібліотека класів .NET Framework - платформа Mono забезпечує повний набір класів, які забезпечують міцну основу для створення додатків. Ці класи сумісні з класами Microsoft .Net Framework.
- Бібліотека Mono Class - Mono також надає багато класів, які виходять за межі бібліотеки базового класу, що надається корпорацією Майкрософт. Вони надають додаткові функціональні можливості, які корисні, особливо при створенні додатків Linux. Деякими прикладами є класи для Gtk +, Zip-файлів, LDAP, OpenGL, Cairo, POSIX і т. Д.

Особливості моно:

- Кросплатформенність – працює на Linux, macOS, BSD та Microsoft Windows, включаючи x86, x86-64, ARM, s390, PowerPC та багато іншого.
- Підтримка багатьох мов – розробка в C# 4.0 (включаючи LINQ і динамічний), VB 8, Java, Python, Ruby, Eiffel, F #, Oxygene та інші.

- Бінарно сумісні – побудовано на впровадженні інфраструктури загальної мови ECMA та C#.
- Microsoft сумісний API – запуск додатків ASP.NET, ADO.NET, Silverlight та Windows.Forms без перекомпіляції
- Відкрите програмне забезпечення, вільне програмне забезпечення.
- Всебічне охоплення технологіями – прив'язки та керовані реалізації багатьох популярних бібліотек та протоколів.

Є багато переваг для вибору Mono для розробки додатків:

- Популярність - Побудований на успіху. Net, є мільйони розробників, які мають досвід створення додатків в C #. Також є десятки тисяч книг, веб-сайтів, підручників та прикладів вихідних кодів, які допоможуть вирішити яку-небудь уявну проблему.
- Програмування на вищому рівні. Всі мови Mono використовують багато функцій середовища виконання, як-от автоматичне керування пам'яттю, відбиття, генерики та різання. Ці функції дозволяють зосередитись на написанні вашої програми замість написання коду системи інфраструктури.
- Бібліотека базового класу - Маючи повноцінну бібліотеку класів, ви знайдете тисячі вбудованих класів для підвищення продуктивності. Необхідний код сокету або хешбайт? Не потрібно писати свої власні, оскільки вони вбудовані в платформу.
- Хрестова платформа - Mono побудована як крос платформа. Моно працює на Linux, Microsoft Windows, MacOS, BSD, Sun Solaris, Nintendo Wii, Sony PlayStation 3, Apple iPhone і Android. Він також працює на x86, x86-64, IA64, PowerPC, SPARC (32), ARM, Alpha, s390, s390x (32 та 64 біти) тощо. Розробка своєї програми з Mono дозволяє працювати майже на будь-якому комп'ютері, який існує.
- Спільна мова виконання (CLR) - CLR дозволяє вам вибрати мову програмування, який вам найбільше подобається, і може взаємодіяти з

кодом, написаним на будь-якій іншій мові CLR. Наприклад, ви можете написати клас у C #, успадкувати від нього в VB.Net, і використовувати його в Eiffel. Ви можете вибрати написання коду в Mono на різних мовах програмування.

2.4.3. Windows forms

Windows Forms (WinForms) - це графічна бібліотека класів (GUI), що входить до складу Microsoft .NET Framework, яка надає платформу для написання багатих клієнтських додатків для настільних ПК, ноутбуків і планшетних ПК. Хоча це розглядається як заміна для більш ранньої та більш складної бібліотеки класів Microsoft Foundation на базі C++, вона не пропонує порівнянних парадигм і діє як платформа для рівня інтерфейсу користувача в багаторівневому рішенні.

Програма Windows Forms - це програма, керована подіями, яка підтримується Microsoft .NET Framework. На відміну від пакетної програми, він витрачає більшу частину свого часу, просто чекаючи, що користувач щось зробить, наприклад, заповнивши текстову панель або натиснувши кнопку.

Windows Forms забезпечує доступ до звичайних елементів керування користувацьким інтерфейсом Windows шляхом обертання існуючого API Windows у керованому коді. За допомогою Windows Forms .NET Framework забезпечує більш повну абстракцію вище API Win32, ніж це робили Visual Basic або MFC.

Форми Windows схожі на бібліотеку Microsoft Foundation Class (MFC) у розробці клієнтських програм. він забезпечує обгортку, що складається з набору класів C++ для розробки додатків Windows. Тим не менш, він не забезпечує роботу за замовчуванням, як MFC. Кожен елемент керування у програмі Windows Forms є конкретним прикладом класу[20].

2.5. Висновки

Розроблена архітектура теоретично має більш швидко опрацьовувати відмови в системі, отже є більш ефективною. Використання приорітетного буфера команд запобігає можливості виникнення ситуації глухого кута, коли без результату даної команди неможна продовжити обчислення.

В запропонованому підході до побудови системи кожній команді виділяється необхідний час її виконання, що прискорює реконфігурацію системи при відмові обчислювальних модулів, тобто зменшує час реалізації заданого алгоритму вирішення задачі.

Оскільки підготовка алгоритму для потокової системи виконується без урахування конкретної кількості ОМ в системі, то існує можливість продовження обчислень доти, поки в системі буде залишатися хоча би один працездатний ОМ. Таким чином досягається динамічна реконфігурація.

Для тестування розробленої архітектури було написано програмний модуль(емулятор), що складаться компілятора псевдомови та графічного поетапного виконання написаної програми. Елементи системи можна конфігурувати за допомогою меню налаштувань, а бібліотка команд, що можуть бути обчислені модулями, з часом їх виконання знаходиться у конфігураційному файлі, котрий можна відредагувати для потреб користувача.

Для написання емулятора було використано кросплатформену платформу розробки Mono, мову C# та Winforms.

3. ПРОГРАМНА МОДЕЛЬ ДЛЯ РОЗРОБЛЕНОЇ АРХІТЕКТУРИ ВІДМОВОСТІЙКОЇ СПД

3.1. Транслятор розробленої псевдомови

Для того щоб проемулювати роботу архітектури та протестувати її ефективність, відмовостійкість та інше необхідно якимось чином представити алгоритм програми для тестування. Для цього було розроблено доволі нескладну псевдомову. Транслятор потрібний для того, щоб транслювати дану псевдомову у граф потоку даних. Такий поділ зроблено для зручності користувачі. Їм не має потреби вникати у стуктуру графів, а просто написати програму звичним будь-якому програмісту чином – послідовним написанням необхідних команд для виконання. Для емуляції даного алгоритму зручніше використовувати саме граф потоку даних. За допомогою нього дуже просто описується основна ідея систем, що керуються потоками даних, тому таким чином її просто емулювати.

Структура розробленого транслятора(рис. 3.1) майже нічим не відрізняється від усіх інших. В основі лежить словник із зарезервованими словами. Різниця полягає в тому, що наш словник може бути розширений необхідними користувачу командами за допомогою конфігураційного файлу. Він заповнюєтсься необхідними командами та часом їх виконання. Транслятор автоматично підвантажить файл, використає необхідні дані для лексичного аналізу та передасть до емулюючої частини час виконання користувацьких команд, необхідний для коректного відображення ефективності розробленої системи.

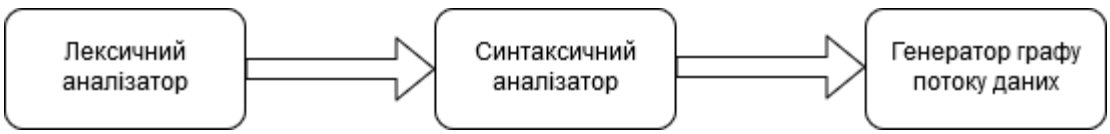


Рисунок 3.1 – Структура транслятора.

Основні складові частини транслятора:

- лексичний аналізатор – переводить псевдокод, зрозумілий користувачу, на набір кодів, що буде зрозумілий компілятору та виконує перевірку лексичних помилок відповідно до стандартного словника слів та його розширення за допомогою конфігураційного файлу;
- синтаксичний аналізатор – перевіряє код на відповідність до синтаксичних правил та створення структурованого формату для генератору;
- генератор – з результатів попередніх аналізів створюється необхідний емулятору гарф потоку даних.

Першим етапом трансляції псевдокоду у граф потоку даних є лексичний аналіз коду. Вхідні дані до цього аналізу:

- вхідний код;
- ключові слова;
- розділові знаки, котрі дозволені в даній мові.

Вхідний код зчитується з файлу, що має розширення .vlm.

Ключові слова складаються з двох основних типів: зарезервовані слова для розуміння семантики та слова, що позначають команди. Розроблена псевдомова має небагато зарезервованих слів:

- PROGRAM – відображає початок головної програми, слідом має бути назва розробленої програми;
- VARS – дане слово означає, що слідом буде список змінних з їх початковим значенням;
- BEGIN – після даного ключового слова починається власне опис алгоритму програми;

- END – означає кінець програми.

Список команд на початку аналізу є порожнім. Він повністю залежить від конфігураційного файлу. Емуляція коду є абстрактною та слугує лише для тестування ефективності та відмовостійкості, тож реалізація логіки конкретних команд не є необхідною. Усі команди є абстрактними і відрізняються лише часом свого виконання. Єдиним виключенням є команда OUT. Вона також відсутня, але обов'язково має бути описана у конфігураційному файлі з часом свого виконання для конкретного користувацького випадку.

Список дозволених розділових знаків:

- , – ставиться між аргументами команди;
- = – ставиться для присвоєння початкового значення змінній;
- [] – між дужками ставиться індекс аргументу для результату;
- _ – ставиться замість аргументу команди, що очікує результату виконання однієї з попередніх команд;
- \n – символ наступного рядка винесений окремо, адже у даній мові він є також символом поділу між командами та присвоєннями.

Окрім вище зазначених знаків дозволені усі пробільні символи, котрі розділяють слова та можуть використовуватися у необмежених кількостях, як буде завгодно користувачу. Також мова передбачає коментарі. Ознакою початку коментаря є комбінація символів (*. Усередині коментаря можливі абсолютно будь-які символи, усе що буде там написано опуститься лексичним аналізатором та не пропуститься на подальший аналіз. Ознакою кінця коментаря є комбінація символів *).

Сам лексичний аналізатор програмно є звичайною машиною станів. Основною одиницею оперування аналізатором є токени. Токен – це логічно завершена частина лексики мови. Це може бути ключове слово, команда, змінна, розділовий знак і тд. Машина станів має всього вісім станів:

- 0 – очікування символу для початку нового токена;
- 1 – у цьому стані поточний токен є ідентифікатором(словом) і очікуваний символ має бути або його продовженням, або кінцем;
- 2 – аналогічний до попереднього стан, різниця полягає лише в тому, що токен є числом;
- 3 – означає, що було зачитано символ (. В цьому стані має прийти символ *, інакше буде згенероване відповідне повідомлення про помилку;
- 4 – аналогічний до станів 2 та 3, токен є розділовим знаком, але так як лексика передбачає лише односимвольні розділові знаки, то очікуваний символ в будь-якому випадку буде сигналізувати про кінець поточного токену;
- 5 – даний стан означає, що попердньо була зафіксована помилка про знаходження літерного символа у числі, для того щоб убезпечитись від непорозумінь даний стан ігнорує всі наступні послідовні літерні символи;
- 6 – стан, що відповідає за обробку коментаря, його функцією є ігнорування усіх символів, окрім *(означає можливе закінчення коментаря);
- 7 – означає, що було зачитано символ * під час коментаря, якщо у даному стані очікуваний символ виявиться), то поточний коментар є завершеним.

При виявлені помилки процес не переривається, а опис проблеми додається до списку. По завершенні роботи машини станів весь список помилок виводиться на екран і вже тоді робота транслятора переривається.

Результатом роботи лексичного аналізатора є список усіх токенів, та словники з виявленими ідентифікаторами та константами. У разі успішного завершення усі ці дані передаються на вхід до синтаксичного аналізатора.

Синтаксичний аналізатор реалізований за допомогою рекурсивного спуску. Синтаксис мови описується наступними правилами:

<vlm-program> => <program>

<program> => PROGRAM <procedure-identifier> \n <block> \n

<procedure-identifier> => <identifier>

<block> => <declarations> BEGIN \n <statements-list> END

<identifier> => <letter><string>

<declarations> => <variable-declarations>

<statements-list> => <statement> <statements-list> | <empty>

<letter> => a..z | A..Z

<string> => <letter> <string> | <number> <string> | <empty>

<variable-declarations> => VARS \n <declaration> <declarations-list> | <empty>

<statement> => <operation-identifier> <variable-identifier> , <variable-identifier> , <unsigned-integer> [<unsigned-integer>] \n

<number> => 0..9

<declaration> => <variable-identifier> = <unsigned-integer> \n

<declarations-list> => <declaration> <declarations-list> | <empty>

<operation-identifier> => <identifier>

<variable-identifier> => <identifier>

<unsigned-integer> => <number> <num-string>

<num-string> => <number> <num-string> | <empty>

Обхід списку токенів виконується за допомогою викликів функцій, котрі відповідають вище описаним правилам та перевіряють чи дотримана потрібна послідовність. Приклад коду, що описує правило <statement>:

```
        private static INode Statement(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants, bool canEmpty = false)
        {
            GrammarNode node = new GrammarNode("<statement>");
            if (!node.AddNode(CheckCommand(lexems, identifiers, "<operation-identifier>",
!canEmpty)))
            {
                if (canEmpty)
                    return new GrammarNode("<empty>");
                return null;
            }
            if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<variable-identifier>",
false)) &&
                !node.AddNode(CheckConstant(lexems, constants, false)) &&
!node.AddNode(Check(lexems, "_")))
            {
                return null;
            }
            if (!node.AddNode(Check(lexems, ",")))
            {
                return null;
            }
            if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<variable-identifier>",
false)) &&
                !node.AddNode(CheckConstant(lexems, constants, false)) &&
!node.AddNode(Check(lexems, "_")))
            {
                return null;
            }
            if (!node.AddNode(Check(lexems, ",")))
            {
                return null;
            }
            if (!node.AddNode(CheckConstant(lexems, constants)))
            {
                return null;
            }
            if (!node.AddNode(Check(lexems, "[")))
            {
                return null;
            }
            if (!node.AddNode(CheckConstant(lexems, constants)))
            {
                return null;
            }
            if (!node.AddNode(Check(lexems, "]")))
            {
                return null;
            }
            if (!node.AddNode(Check(lexems, "\n")))
            {
                return null;
```

```
        }
        return node;
    }
```

Як видно з вище поданого коду, функція складається з послідовних перевірок вхідних токенів на відповідність правилу. У випадку коректності будується дерево розбору, що і є результатом роботи алгоритму. Якщо ж ні, то робота переривається відразу з виведенням опису виявленої проблеми. Якщо помилок не було виявлено, пройдено всі правила та не залишилося не проаналізованих токенів, то вважається, що розбір був успішним.

Дерево розбору з синтаксичного аналізатора та словники з лексичного ідуть на вхід до останнього етапу трансляції – генерації графу потоку даних.

Генератор проходить по дереву розбору та перевіряє семантичні помилки. Наприклад, щоб ідентифікатор змінної не співпадав з ідентифікатором програми, команди або ключовим словом. Проходячи по деклараціям змінних, генератор заповнює внутрішню базу змінних з їх початковими значеннями. Потім, проходячи по командам, формується власне граф потоку даних, при цьому використовуються словники з лексичного аналізатора та внутрішній генератора з деклараціями.

3.2. Емуляційна частина

Частина програмної моделі, що відповідає за емуляцію складається з чотирьох основних модулів:

- Головний модуль програми, що визначає інтерфейс з користувачем і слугує для вводу та обробки алгоритмів та даних.
- Модуль, що відповідає за емуляцію роботи середовища формування команд.

- Модуль, що надає можливість конфігурувати параметри системи (кількість процесорних елементів, пристроїв вводу та виводу даних, розміри пам’яті, параметри набору команд процесорних елементів).
- Модуль формування системи команд та створення бібліотеки команд, що може виконувати обчислювальний модуль.

Схема їх комунікації зображена на рисунку 3.2.

Конфігураційний модуль надає можливість досить гнучко налаштувати систему користувачу.

Даний модуль надає можливість обрати:

- кількість обчислювальних модулів, що будуть у модельованій системі;
- кількість вхідних пристроїв системи;
- кількість вихідних пристроїв системи;
- розміри пріоритетної та звичайної буферної пам’яті команд;
- розмір доступної асоціативної пам’яті;
- швидкість доступу до усіх типів пам’яті.

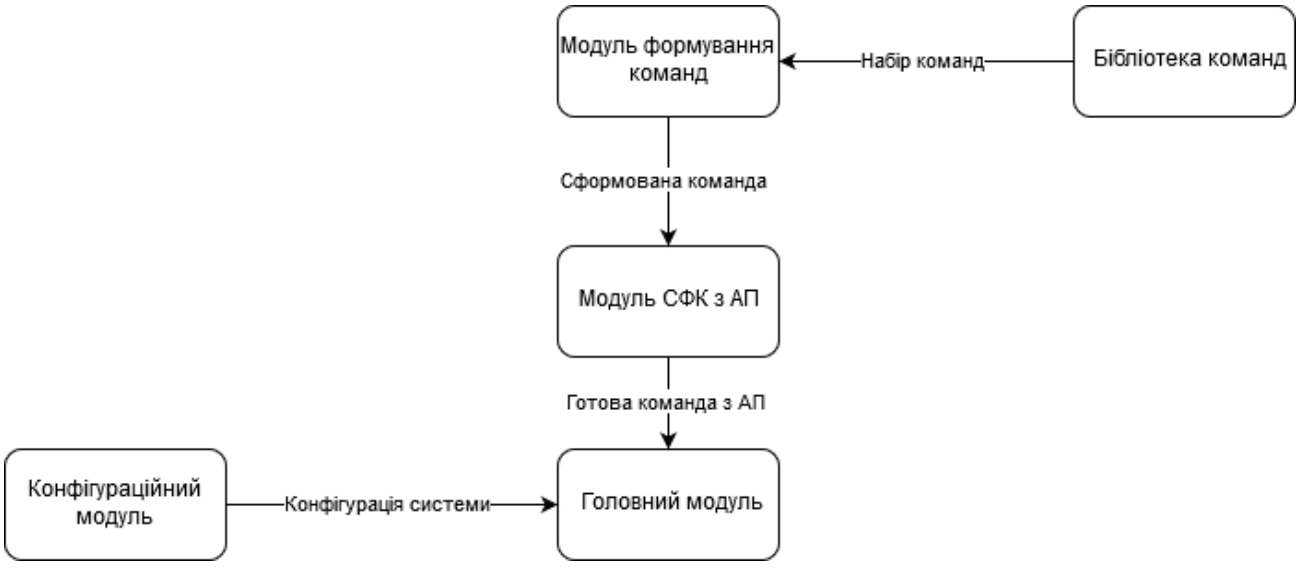


Рисунок 3.2 – Структура емуляційної частини програмної моделі.

Бібліотека команд з часом їх виконання надходить з конфігураційного файлу. Основною всієї емуляції граф потоку даних, що формується з коду програми та надходить з транслятора, описаного у попередньому підрозділі. Ці дані є необхідними для коректної оцінки ефективності. Усі команди є абстрактними, а часові відрізки відносними.

Загальна структура програмної моделі дуже схожа на структуру розробленої архітектури системи, що керується потоком даних(рис. 3.3). Окрім основних модулів, зазначених вище, існують менші модулю, котрі відповідають елементам архітектури. Одним з них є модуль вхідних пристроїв.

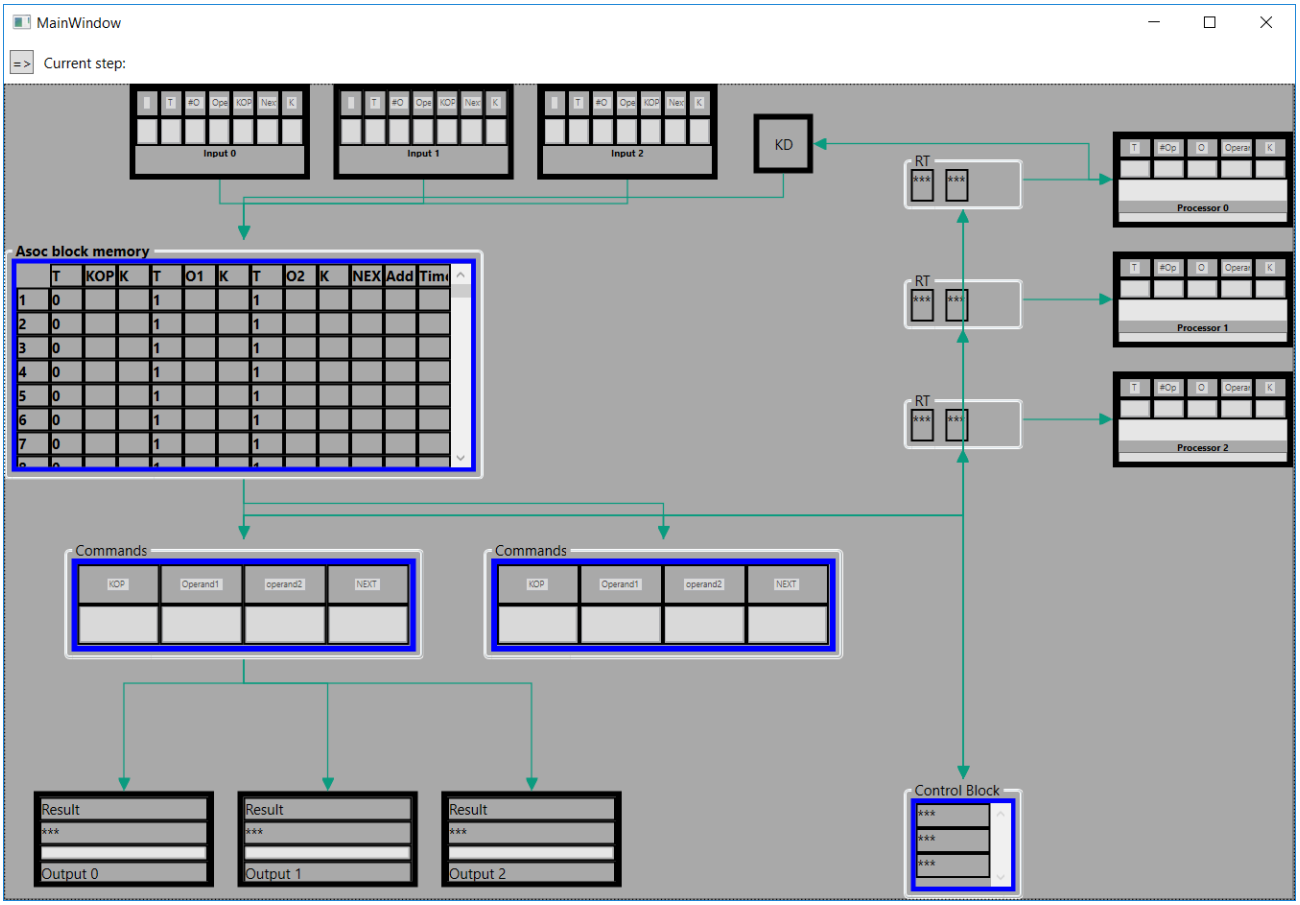


Рисунок 3.3 – Головний екран емулятору.

Модуль вхідних пристроїв емулює роботу першого комунікаційного середовища. Саме цей модуль виконує розбір графу потоку даних. На рисунку 3.4 видно, що відразу ж на першому кроці модуль дістав перші дані з графу та заповнив в усі вхідні пристрої: input0 – input2.

Модуль формування системи команд приймає дані, що вже пройшли вхідні пристрої та записує їх у відповідне місце в асоціативній пам’яті. Для цього необхідно створити відповідний об’єкт класу Word, що змістом має ті самі поля, що слова, описані в попередньому розділі. Від цього класу наслідується ще два: Actor та Operand. Саме їх екземпляри і формуються у модулі формування системи команд.

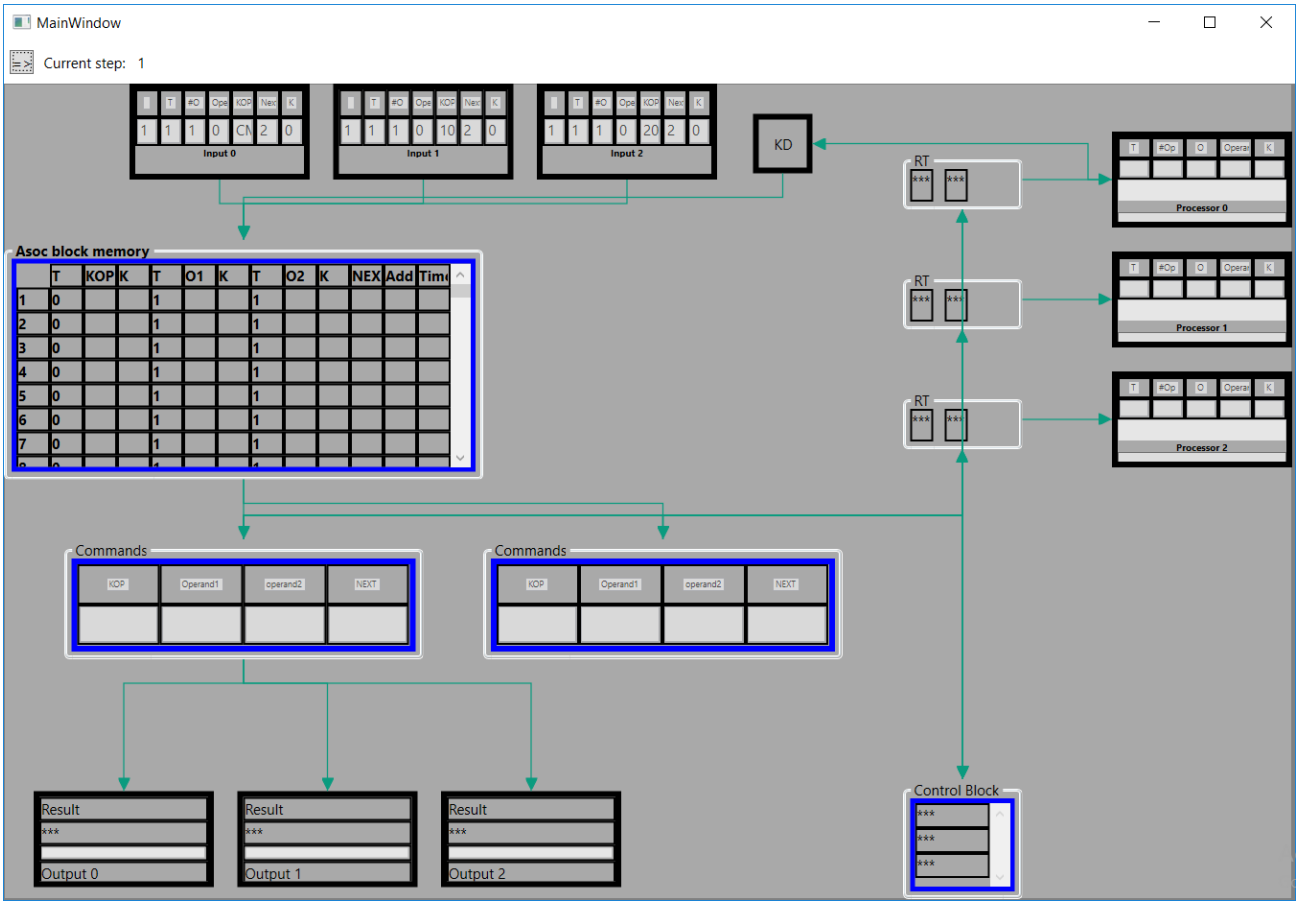


Рисунок 3.4 – Заповнення вхідних пристроїв.

Асоціативна пам’ять в емуляторі представлена списком об’єктів типу AsocElem. Даний об’єкт містить інформацію про актора(class Actor), його операнди(class Operand), їх ознаки та інші додаткові поля. В результаті роботи модуля формування системи команд дана пам’ять і заповнюється(рис. 3.5).

Модуль середовища формування команд слідкує за перебігом справ в асоціативній пам’яті, а точніше за ознаками елементів цього списку. Коли ознаки, що відповідають актору та обом операндам однієї ж структури в списку асоціативної пам’яті дорівнюють одиничкам(це означає, що всі елементи команди готові), то дана команда, разом зі своєю адресою записується до буферної пам’яті команд, що програмно представлена списком об’єктів типу BufElem(рис. 3.6).

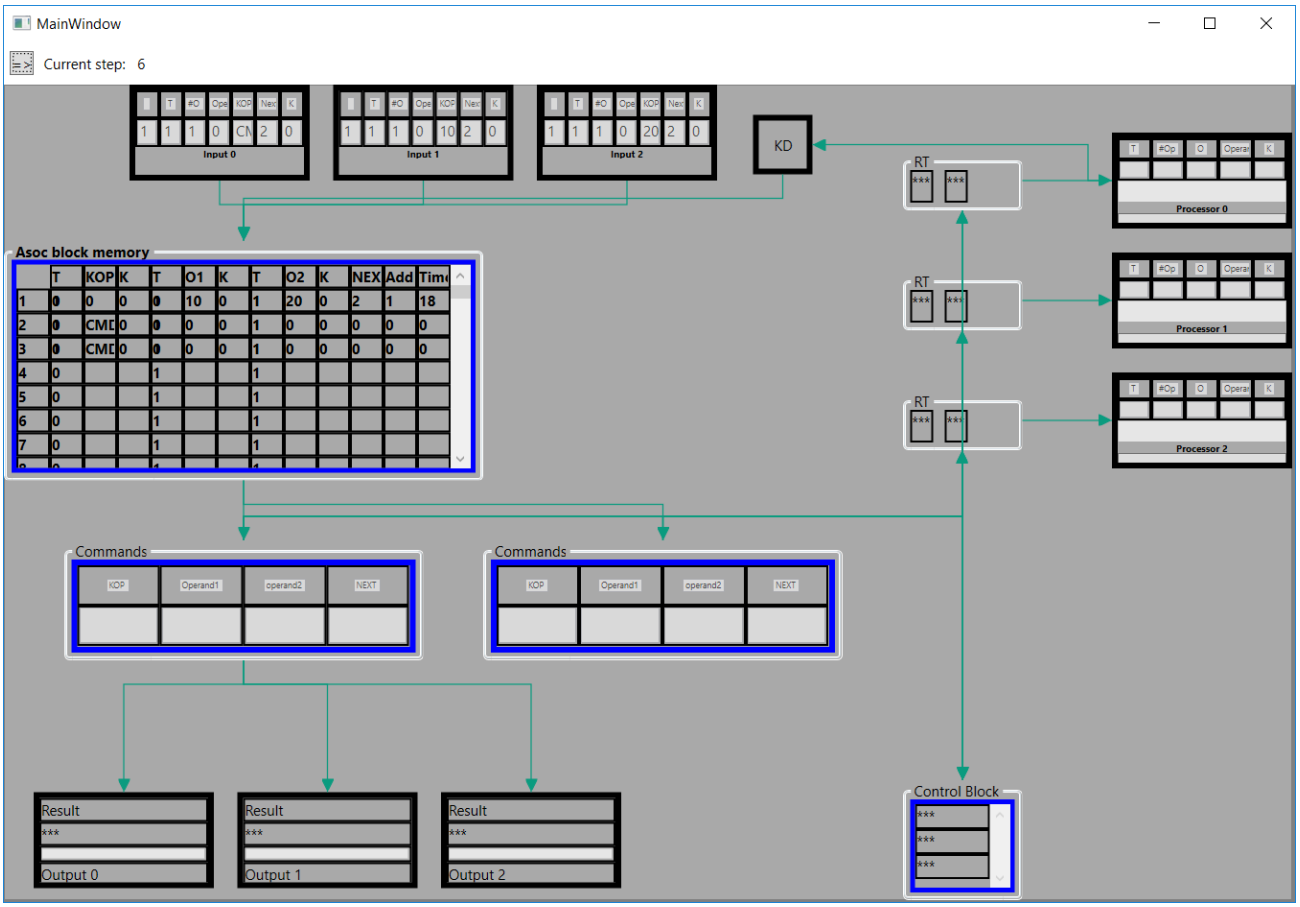


Рисунок 3.5 – Заповнення асоціативної пам’яті.

На цьому етапі вступає головний модуль програми. По-перше він слідкує за буферною пам’яттю команд. Коли готова команда потрапляє до буферної пам’яті, то головний модуль починає шукати вільний обчислювальний модуль для початку її виконання. Першочергово перевіряється пріоритетний буфер пам’яті. Коли вільний процесор знайдено, то у відповідну комірку масиву, що відповідає тимчасовим регістрам записується адреса команди з асоціативної пам’яті. Сама ж команда передається до модуля, що відповідає роботі обчислювальних модулів та таймерів. По-друге головний модуль слідкує за станом обчислювальних модулів.

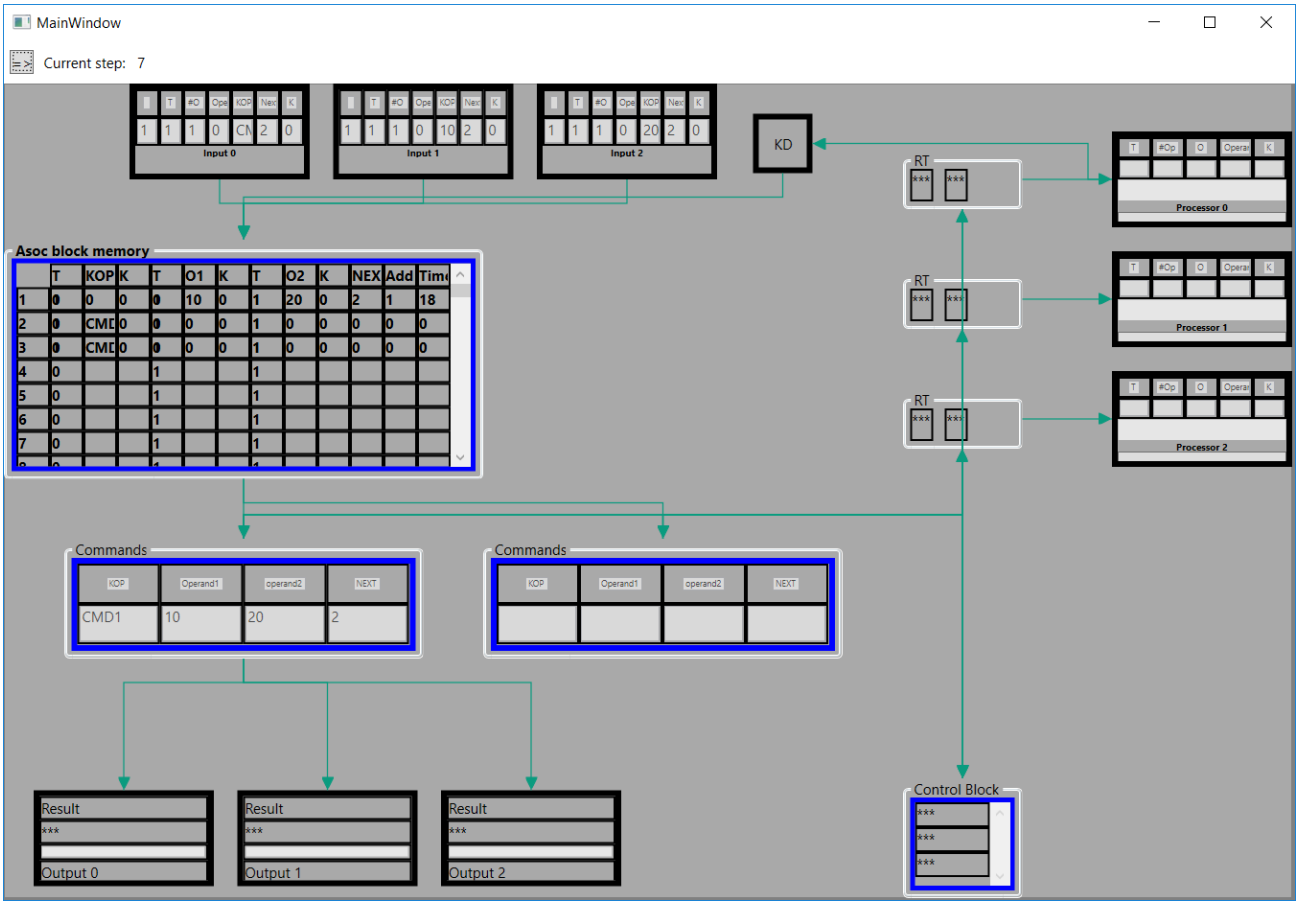


Рисунок 3.6 – Заповнення буферної пам’яті.

Модуль, що відповідає роботі обчислювальних модулів та таймерів після отримання команди та номеру процесора, на якому вона має виконатися, запускає відповідний таймер та робить відповідні графічні зміни на головному екрані. Також, з кожним кроком користувача, даний модуль оновлює таймери та необхідну інформацію(рис 3.7).

У разі, якщо користувач захоче перевірити відмовостійкість, він може вимкнути один з обчислювальних модулів(рис. 3.8). У цьому випадку головний модуль дістане адресу команди з масиву, що відповідає тимчасовим регістрам, дістане за цією адресою команду в асоціативній пам’яті та запише її у пріоритетний буфер команд(рис. 3.9). Надалі все виконується за тим же алгоритмом, що і раніше. У разі відновлення користувачем роботи несправного обчислювального модуля, то він лиш помічається головним модулем, як справний. На алгоритм це не впливає жодним чином.

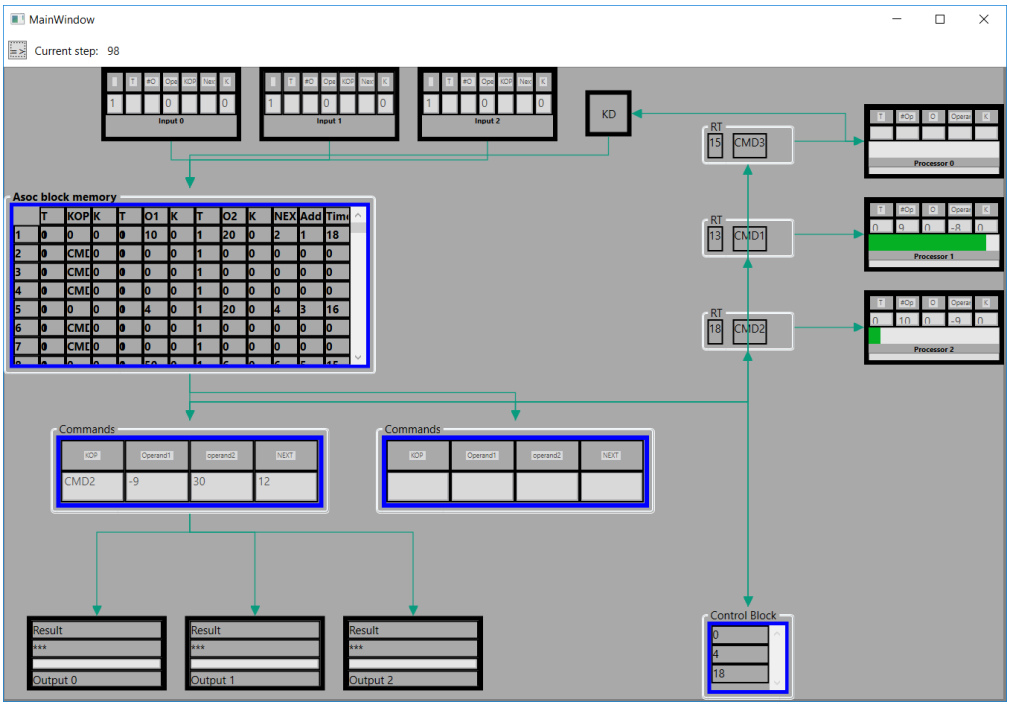


Рисунок 3.7 – Робота емулятора.

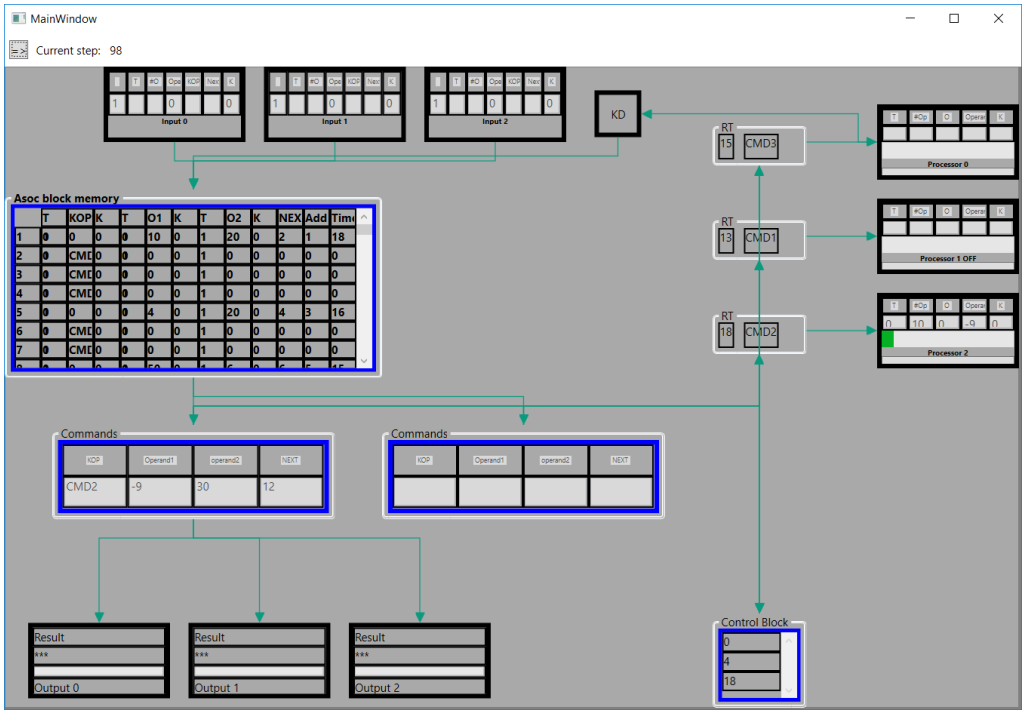


Рисунок 3.8 – Вимкнення другого обчислювального модуля.

Командою OUT активується початок процесу виведення інформації через пристрої виведення. Це виконує модуль виведення, що відповідає другому комунікаційному середовищу(рис. 3.10).

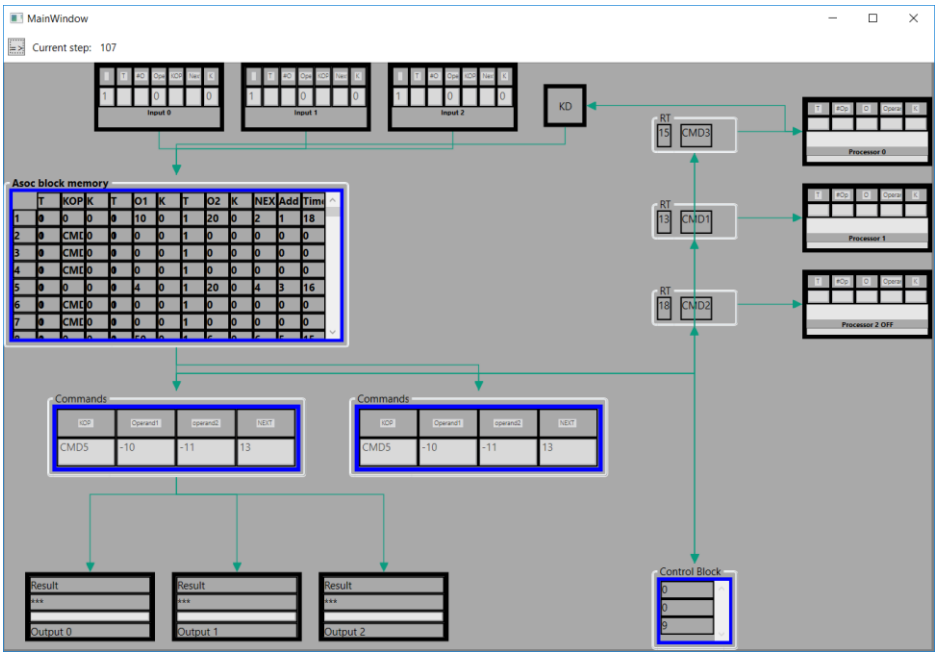


Рисунок 3.9 – Запис до пріоритетного буфера команд.

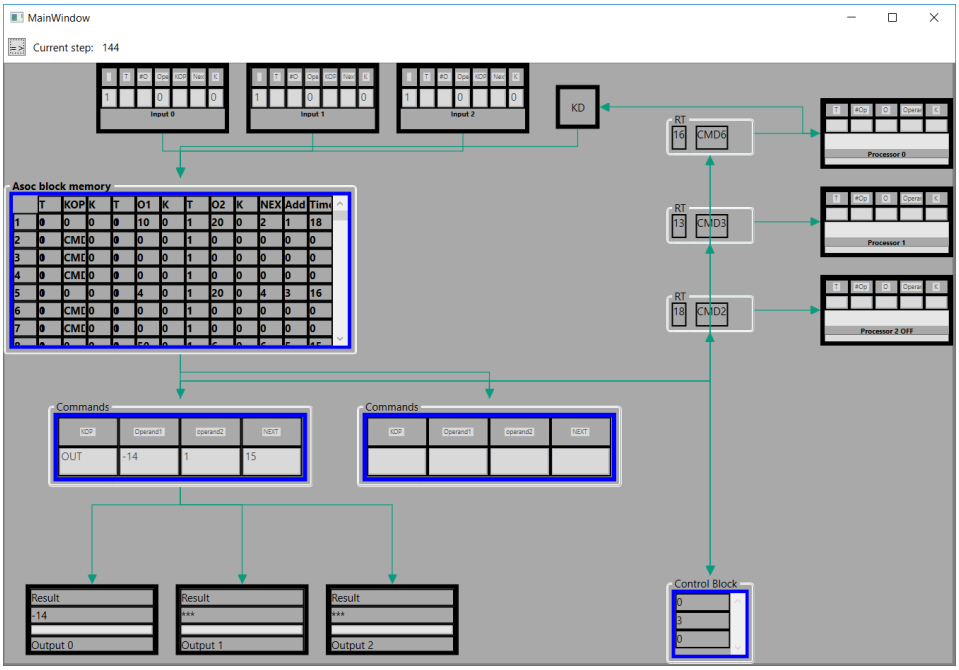


Рисунок 3.10 – Запис до вихідних пристроїв.

3.3. Висновки

У даному розділі було описано розроблену програмну модель. Докладно прояснено роботу усіх частин транслятора та емулятора. Наведено частини коду та скріншоти візуальної частини програмної моделі. Навіть без докладного аналізу результатів, лише з даного опису алгоритму, вже можна стверджувати, що було досягнуто динамічну реконфігурацію. Архітектура не залежить від кількості працюючих обчислювальних молулів. Для роботи всієї системи достатньо щоб був працюючий хоча б один.

Розроблений транслятор надає зручний спосіб введення алгоритму користувачем та в результаті надає граф потоку даних для простої обробки в емуляційній частині програмного модуля. Емуляційна частина, в свою чергу, надає користувачу гнучкі можливості конфігурації системи, достатньо просту та зрозумілу візуальну частину, можливості покрокового виконання алгоритму для

тестування ефективності, вимикання та вмикання обчислювальних модулів для перевірки відмовостійкості.

4. АНАЛІЗ РЕЗУЛЬТАТІВ РОБОТИ ВІДМОВОСТІЙКОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ, ЩО КЕРУЄТЬСЯ ПОТОКОМ ДАНИХ

4.1. Порівняльний аналіз ефективності існуючих систем з розробленою на прикладі симеричного блочного алгоритму шифрування даних IDEA.

Для коректної та об’єктивної оцінки існуючих відмовостійких обчислювальних систем було додатково розроблено емулятори для систем, котрі будуть використані для порівняння. Дані емулятори були розроблені на основі програмної моделі, що пропонується. Для порівняння були обрані наступні архітектури:

- Розроблена архітектура з використанням асоціативної пам’яті (рис. 4.5);
- Існуюча обчислювальна система з використанням асоціативної пам’яті (рис. 4.6);
- Існуюча обчислювальна система з використанням пам’яті з довільним адресним доступом(ПДД) (рис. 4.7).

Для більш точного аналізу було розглянуто декілька можливих варіантів поведінки систем:

- N обчислювальних модулів весь час працюють справно;
- Через деякі проміжки часу обчислювальні модулі по одному починають відключатися;
- Коли залишається лише один справний обчислювальний модуль, то по одному вони починають відновлюватися.

IDEA використовує 128-бітний ключ і 64-бітний розмір блоку. Вхідний текст розбивається на блоки по 64 біт. Якщо таке розбиття неможливо, останній блок доповнюється різними способами певною послідовністю біт. Для уникнення витоку інформації про кожен окремий блок використовуються різні режими

шифрування. Кожен вихідний незашифрований 64-бітний блок ділиться на чотири підблока по 16 біт кожен, так як всі операції алгебри, що використовуються в процесі шифрування, відбуваються над 16-бітними числами. Для шифрування і розшифрування IDEA використовує один і той же алгоритм.

Зі 128-бітного ключа для кожного з восьми раундів шифрування генерується по шість 16-бітних підключів, а для вихідного перетворення генерується чотири 16-бітних підключа. Всього буде потрібно 52 = 8 x 6 + 4 різних підключів по 16 біт кожен. Процес генерації п'ятдесяти двох 16-бітних ключів полягає в наступному:

- Насамперед, 128-бітний ключ розбивається на вісім 16-бітових блоків. Це будуть перші вісім підключів по 16 біт кожен - ($K_1^{(1)}$, $K_2^{(1)}$, $K_3^{(1)}$, $K_4^{(1)}$, $K_5^{(1)}$, $K_6^{(1)}$, $K_1^{(2)}$, $K_2^{(2)}$).
- Потім цей 128-бітний ключ циклічно зсувається вліво на 25 позицій, після чого новий 128-бітний блок знову розбивається на вісім 16-бітових блоків. Це вже наступні вісім підключів по 16 біт кожен - ($K_3^{(2)}$, $K_4^{(2)}$, $K_5^{(2)}$, $K_6^{(2)}$, $K_1^{(3)}$, $K_2^{(3)}$, $K_3^{(3)}$, $K_4^{(3)}$).
- Процедура циклічного зсуву і розбивки на блоки триває до тих пір, поки не будуть згенеровані всі 52 16-бітових підключа.

Над 16-бітними підключами і підблоками незашифрованого тексту виконуються наступні операції:

- множення по модулю 65537, причому замість нуля використовується 65536 $\odot$;
- додавання по модулю 65536 $\boxplus$;
- побітовое виключне АБО $\oplus$.

Графічно один крок алгоритму відображений на рисунку 4.1.

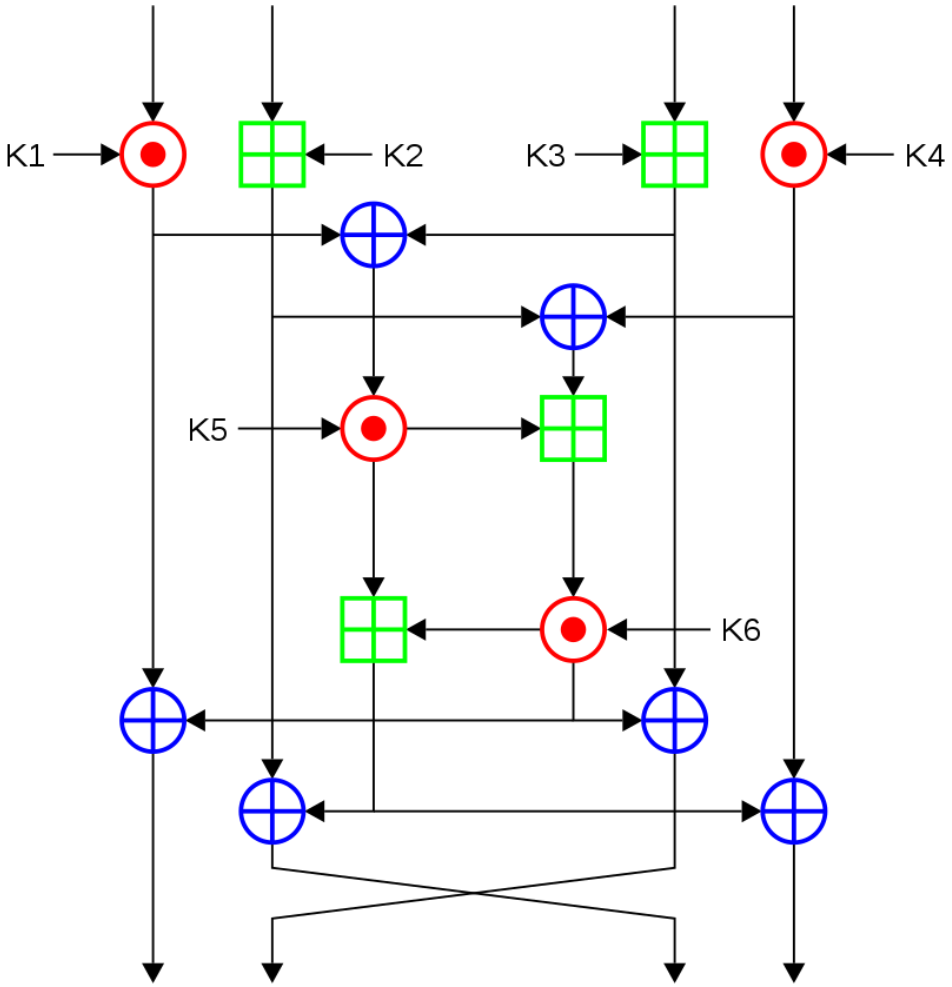


Рисунок 4.1 – Крок кодування алгоритму IDEA

Вхідні дані емулятору:

- 128-бітний ключ K = $0001000200030004000500060007000{8}_{16}$;
- Вхідне значення D = 3.

Для одного вхідного значення(слова) виконається такий набір операцій:

- множення по модулю 65537 – 4 * 8(к-сть кроків) + 2(вихідний крок) = 36;
- додавання по модулю 65536 – 4 * 8(к-сть кроків) + 2(вихідний крок) = 36;
- побітовое виключне АБО – 6 * 8(к-сть кроків) + 0(вихідний крок) = 48.

Було змодельовано архітектури, що мають від одного до десяти обчислювальних модулів.

Результати у зведених таблицях 4.1, 4.2, 4.3.

Таблиця 4.1 – Всі обчислювальні модулі справні.

Обчислювальна система	***ОМ***	***К-сть умовних тактів***
Розроблена ОС	1	2341
	2	1573
	3	1163
	4	849
	5	712
	6	687
	7	604
	8	526
	9	487
	10	456
ОС з АП	1	2360
	2	1683
	3	1201
	4	893
	5	734
	6	703
	7	643
	8	584
	9	523
	10	501
ОС з ПДД	1	2374
	2	1703
	3	1242
	4	917
	5	774
	6	725
	7	681
	8	600
	9	547
	10	519

З таблиці 4.1 видно, що серед вже існуючих методів, в ефективності переважає обчислювальна система з асоціативним доступом до пам’яті, що вже вказує на доцільність її використання при відмовостійких системах. Розроблена архітектура з асоціативною пам’яттю, яка є покращеною версією вже існуючої, показала свою перевагу навіть при усіх справних обчислювальних модулях. Коли кількість ОМ стає більшою за деяке значення(у даному випадку 5-6 процесорів), то в усіх системах спостерігається вже не такий значний приріст у швидкості ніж при менших кількостях. Дана інформація може стати доволі корисною при проектуванні реальних(фізичних, апаратних) систем. Зрозуміло, що це залежить від набору операцій, потужності обчислювальних модулів та розробленого програмного забезпечення.

Наступним етапом стала перевірка відмовостійкості. Для цього випадку був використаний той самий код алгоритму шифру IDEA, що був проаналізований вище. Проте на цей раз експерименти починаються з роботи усіх десяти обчислювальних модулів та поступового відключення по одному з них через деякий, заздалегідь підібраний, час. У даному разі процесори вимикались кожні 50 тактів. Цей експеримент дає змогу перевірити відмовостійкість обчислювальних систем. З результатів можна проаналізувати чи продовжує система свою роботу та наскільки швидко вона відновлюється після відключення процесору. Дана можливість реалізована у інтерфейсі емулятора.

Таблица 4.2 – Поступове відключення обчислювальних модулів.

Обчислювальна система	***Несправні ОМ***	***К-сть умовних тактів***
Розроблена ОС	1	467
	2	501
	3	578
	4	641
	5	701

	6 7 8 9	804 989 1356 1935
ОС з АП	1 2 3 4 5 6 7 8 9	517 551 622 675 723 803 1023 1465 1947
ОС з ПДД	1 2 3 4 5 6 7 8 9	533 579 645 704 743 834 1123 1563 2152

Проаналізувавши результати з таблиці 4.2 видно, що система, котра використовує пам’ять з довільним адресним доступом при відмовах обчислювальних модулів стає значно повільнішою. Це пов’язано з досить швидким доступом до асоціативної пам’яті при повторному виконанні команди, що не виконалась на несправному обчислювальному модулі. При порівнянні двох архітектур з асоціативною пам’яттю перевагу має саме розроблений алгоритм, котрий модифікував роботу з таймерами. Проте є випадки, коли це може бути не вигідно. Наприклад, при відмові шести процесорів ефективність вже існуючої обчислювальної системи трохи краща, але в загальному випадку це є мізерною різницею.

В наступному експерименті перевірятиметься поведінка систем при відновленні відмовлених процесорів. У розробленому емуляторі було передбачено і такий інтерфейс. Відновлення обчислювальних модулів

відбуватиметься з тим самим періодом, що й вимикання(50 тактів). В усіх нижче описаних випадках спочатку поступово відмовляють усі процесори(крім останнього), а потім поступово починаються по одному вмикатися.

Таблица 4.3 – Поступове відновлення відмовлених обчислювальних модулів.

Обчислювальна система	***Відновлені ОМ***	***К-сть умовних тактів***
Розроблена ОС	1	1674
	2	1124
	3	920
	4	754
	5	689
	6	623
	7	546
	8	489
	9	456
ОС з АП	1	1748
	2	1245
	3	948
	4	759
	5	703
	6	654
	7	589
	8	531
	9	502
ОС з ПДД	1	1823
	2	1342
	3	924
	4	798
	5	725
	6	670
	7	603
	8	553
	9	524

Всі, пророблені за допомогою емулятора вище описані випадки, були зведені в одну таблицю 4.4 для порівняння.

Таблиця 4.4 – Підсумковий аналіз опрацьованих експериментів.

Умови експерименту	***Відсотки від ОС з АП***	***Відсотки від ОС з ПДД***
Справні ОМ	0.99 0.93 0.97 0.95 0.97 0.98 0.94 0.90 0.93 0.91	0.99 0.92 0.94 0.93 0.92 0.95 0.89 0.88 0.89 0.88
Несправні ОМ	0.90 0.91 0.93 0.95 0.97 1.00 0.97 0.93 0.99	0.88 0.87 0.90 0.91 0.94 0.96 0.88 0.87 0.90
Відновлені ОМ	0.96 0.90 0.97 0.99 0.98 0.95 0.93 0.92 0.91	0.92 0.84 1.00 0.94 0.95 0.93 0.91 0.88 0.87

За зведеною таблицею 4.4 гарно видно, що у відсотковому співвідношенні розроблений алгоритм працює не гірше ніж існуючі, а в більшості випадків навіть краще. При стабільній роботі ОС з асоціативною пам’яттю значно випереджають варіант з довільним адресним доступом. У разі відмов спостерігається та сама картина. Проте, відновлення обчислювальних модулів проходить ефективніше

саме при системі з довільним доступом до пам’яті. Графічно результати продемонстровані на рисунках 4.2, 4.3, 4.4.

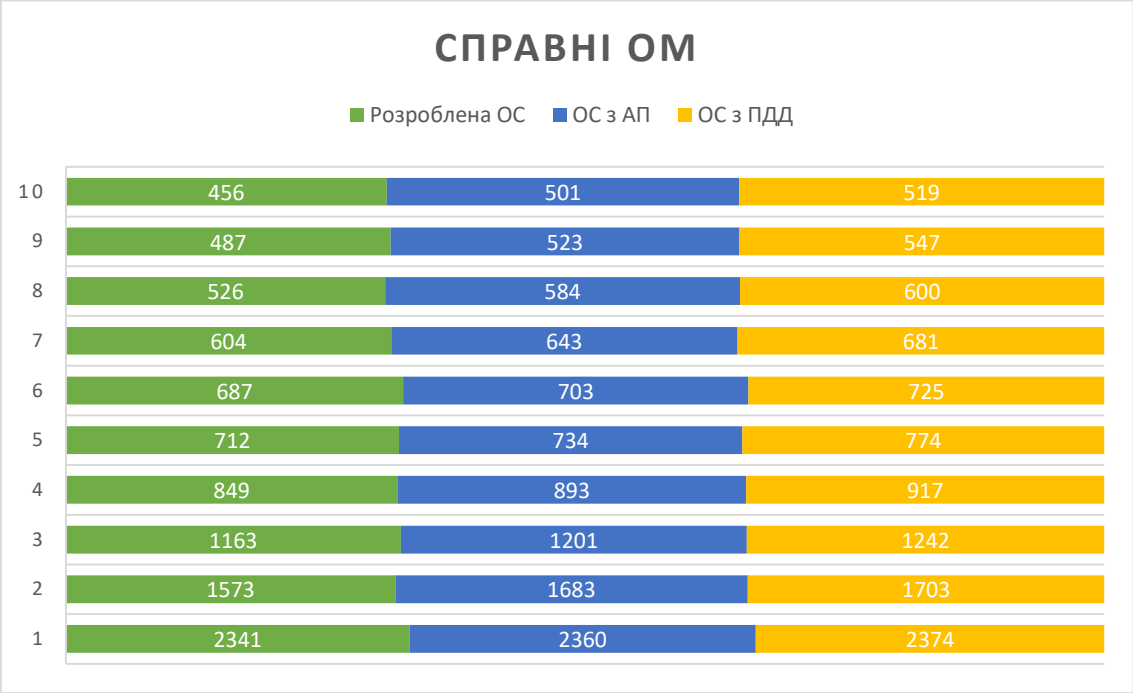


Рисунок 4.2 – Справні ОМ.

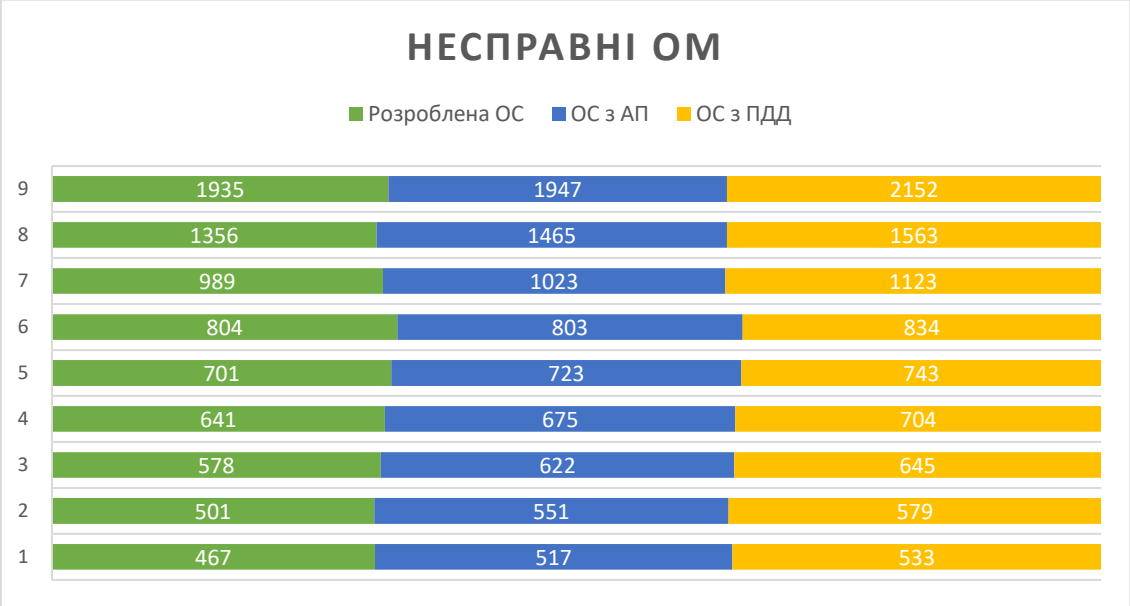


Рисунок 4.3 – Несправні ОМ.

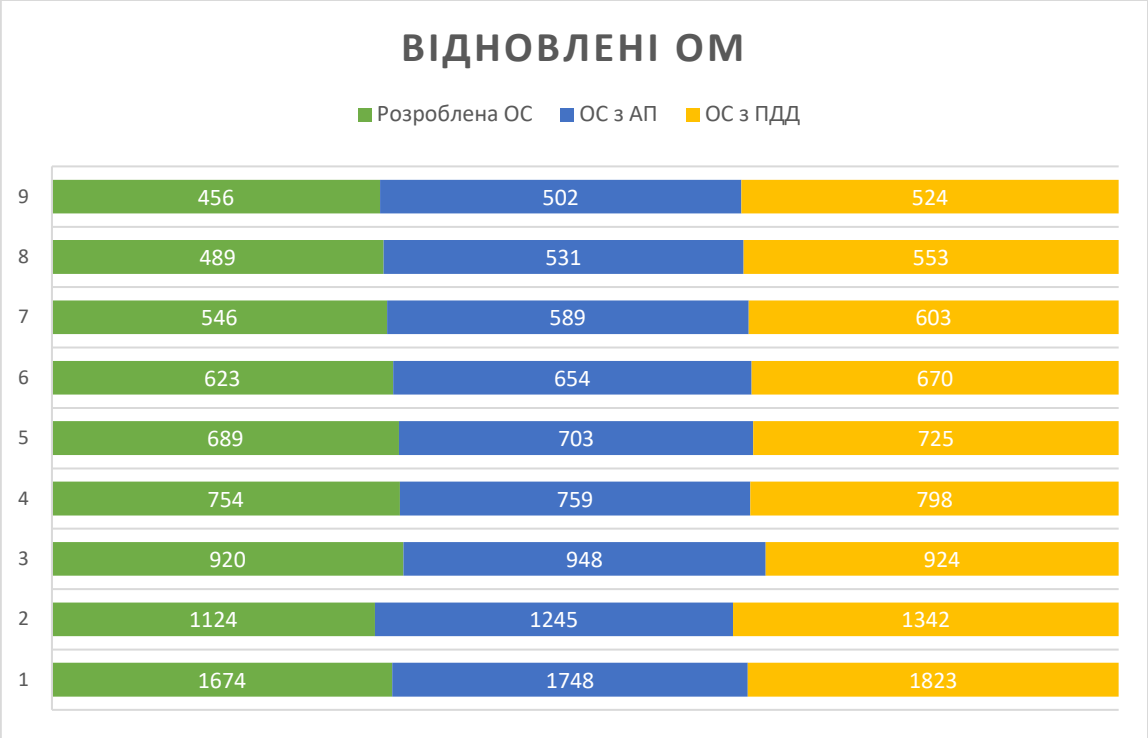


Рисунок 4.4 – Відновлені ОМ.

За допомогою діаграм чітко видно, що розроблена система займає найменшу кількість часу. Хоча виміри мають лише умовний характер, проте для відносних порівнянь цього цілком достатньо і можна робити деякі висновки. У подальшому в налаштуваннях емулятора можна буде задати необхідні параметри обчислювальних модулів: швидкісні характеристики, набори команд та швидкість їх виконання та інше. За бажанням користувач може використовувати необхідні йому часові рамки для замірів.

У висновках можна відзначити, що розроблена система не лише виявилася достатньо ефективною, щоб не програти існуючим, а й на відмінно впоратися з відмовами обчислювальних модулів та їх відновленням. При цьому майже не втрачаючи в швидкостях, хіба що на етапах відновлення. Проте ці затрати

нівелюються швидким доступом при відмовах до асоціативної пам’яті та модифікованим варіантом використання таймерів.

4.2. Аналіз ефективності розробленої відмовостійкої обчислювальної системи з урахування особливостей роботи її окремих компонентів

Незважаючи на загальну модель обчислень, потокові системи мають суттєві відмінності, пов'язані зі способом активізації команд. Існують різні підходи до структурної організації СФК. Відомі методи формування команд в СФК на основі асоціативної пам'яті і пам'яті з довільним адресним доступом до комірок. У потокових системах підготовка задачі здійснюється без урахування кількості обчислювальних модулів. Це створює передумови в разі відмови ОМ продовжувати обчислення до тих пір, поки в системі не залишиться хоча б один працездатний ОМ. З огляду на те, що методи забезпечення відмовостійкості пам’яті розроблені досить добре, основна увага приділяється апаратним засобам автоматичної реконфігурації системи при відмові ОМ. При цьому основною проблемою є відновлення інформації про команду, загубленої в зв'язку з відмовою ОМ.

Нехай для визначеності ОМ виконують двохадресні операції (для одноадресних може бути введений фіктивний операнд). Тоді команди з акторів $A = \langle I, F, N, P \rangle$ і даних $D = \langle I, Q, N, P \rangle$ формуються в форматі $\langle A, D_1, D_2 \rangle$. В даному підрозділі буде детальніше описано про розроблену та обрані для порівняння архітектури систем.

Схеми відповідних архітектур наведені на рисунках 4.5, 4.6 та 4.7.

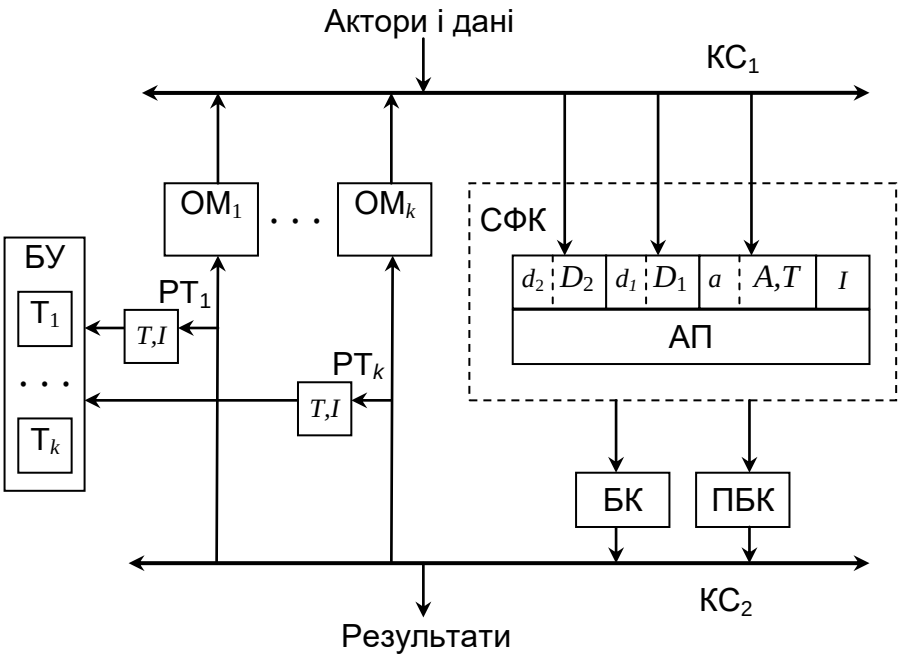


Рисунок 4.5 – Розроблена обчислювальна система з використанням асоціативної пам’яті.

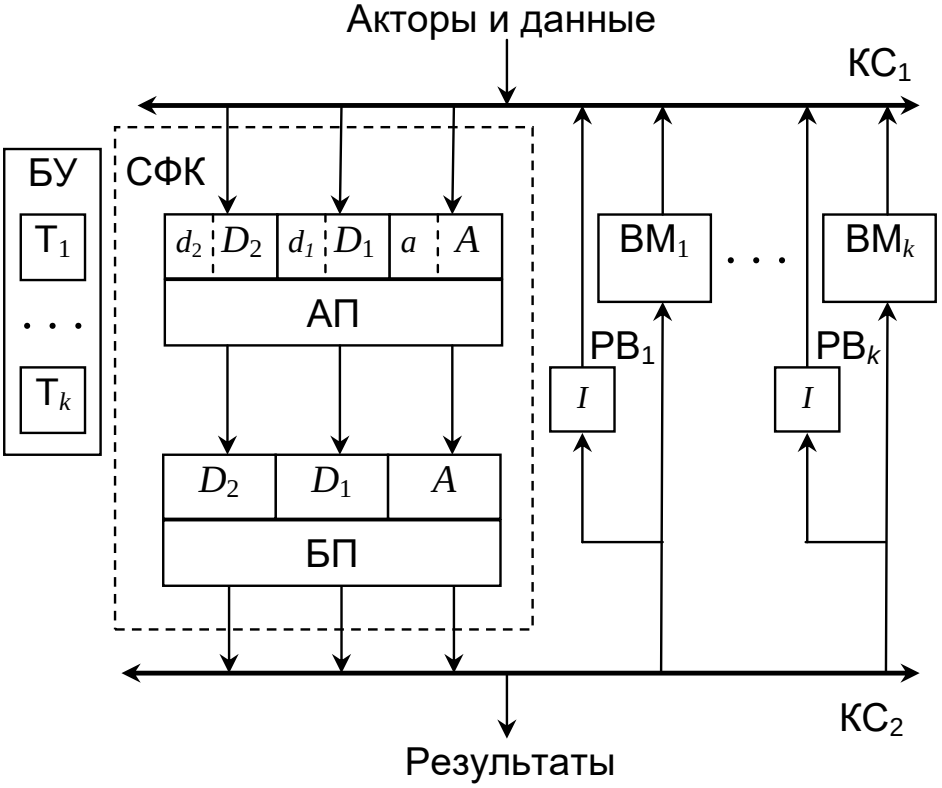


Рисунок 4.6 – Існуюча обчислювальна система з використанням асоціативної пам’яті.

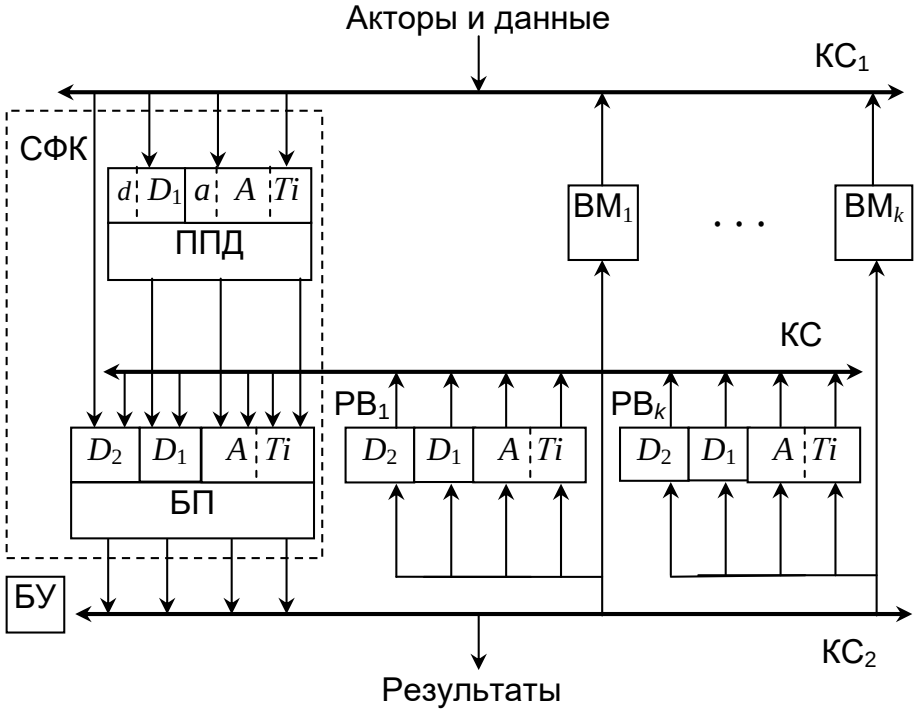


Рисунок 4.7 – Існуюча обчислювальна система з використанням пам’яті з довільним адресним доступом.

1.Розроблена обчислювальна система (рис 4.5).

У склад системи входять обчислювальні модулі (ОМ), середовище формування команд (СФК) на базі асоціативної пам’яті (АП), регістри для тимчасового зберігання інформації (РТ), блок управління (БУ), буферна пам’ять команд (БК) та комутаційні середовища (КС).

В даній роботі розглядається наступний підхід до забезпечення відмовостійкості СПД. Для визначення несправностей ОМ використовується контроль часових інтервалів. Контролюючим об’єктом є апаратно захищене ядро системи, під яким розуміють СФК та БУ.

Формування команд відбувається наступним чином. Актори і дані (токени) в будь-якому порядку поступають через КС1 в СФК. З використанням процедури адресного запису актори і дані для i –ї команди записуються в одну комірку АП

за адресою I_i, що співпадає для всіх об'єктів одної команди. Одночасно із записом актора і даних встановлюються ознаки їх наявності у відповідних розрядах комірки пам'яті (d_2, d_1, a). Значення ознак $d_2 d_1 a = 111$ вказує на наявність в АП готової команди для ОМ у формі

$$I : \langle D_2; D_1; A.T \rangle,$$

де I – адреса комірки АП, D_2, D_1 – дані; A,T – актор (код операції) з тривалістю виконання команди.

Готова команда зчитується з АП за допомогою процедури асоціативного читання і переписується до БК типу FIFO. Цикл безадресного читання даних з БК є значно меншим циклу асоціативного читання з АП, що дає можливість прискорити обчислення.

Вільний ОМ_j приймає команду з БК. Одночасно в РТі приймається адреса команди та тривалість відповідного інтервалу її виконання. Код ініціалізації інтервалу переписується у відповідний таймер БУ. Число таймерів у БУ має бути рівним кількості ОМ. Адреса I_i на час виконання команди зберігається в РТ_j. Одночасно з цим зкидуються у нульовий стан ознаки у відповідній комірці АП і блокується запис нової інформації у вказану комірку. Доки ознаки знов не будуть встановлені в одиницю, команда вважається неготовою і не може бути виконана повторно іншим ОМ. Разом з цим у БУ запускається таймер Т_j і починається виконання команди.

Після успішного виконання команди результат через КС1 записується в АП. Разом з цим встановлюється у нуль таймер Т_j та розблоковується адресний запис у комірку з адресою I_i, тобто ця комірка може бути використана для формування іншої команди із відповідним ім'ям (адресою).

У разі відмови ОМ_j спрацьовує таймер Т_j (закінчується ліміт часу на виконання команди). Команда активується шляхом встановлення ознак $d_2 d_1 a = 111$ у комірці АП з адресою I_i, яка зберігалася в RT_j. Ця готова команда записується у пріоритетний буфер команд ПБК. Вільний ОМ зчитує команду з ПБК і вона виконується повторно, як вказано вище. При передачі команд до ОМ спочатку перевіряється наявність команд у ПБК, що запобігає можливості виникнення ситуації глухого кута, коли без результату даної команди неможна продовжити обчислення.

В запропонованому підході до побудови системи кожній команді виділяється необхідний час її виконання, що прискорює реконфігурацію системи при відмові обчислювальних модулів, тобто зменшує час реалізації заданого алгоритму вирішення задачі.

Оскільки підготовка алгоритму для потокової системи виконується без урахування конкретної кількості ОМ в системі, то існує можливість продовження обчислень доти, поки в системі буде залишатися хоча би один працездатний ОМ.

2.Існуюча обчислювальна система з асоціативною пам'яттю (рис 4.6).

До складу системи входять обчислювальні модулі (ОМ), середовище формування команд (СФК) на базі асоціативної пам'яті (АП), регістри для тимчасового зберігання інформації (РВ), блок управління (БУ), буферна пам'ять (БП) і комутаційні середовища (КС).

Формування команд проводиться наступним чином. Актори і дані в будь-якому порядку надходять через КС1 в СФК. З використанням процедури адресного запису актори і дані для і-й команди записуються в одну клітинку АП за адресою I_i, яка збігається для всіх об'єктів однієї команди. Одночасно з записом актора і даних встановлюються признаки їх наявності у відповідних розрядах комірок пам'яті (d_2, d_1, a). Значення ознак $d_2 d_1 a = 111$ вказує на наявність в АП

готової команди для ОМ. Готова команда зчитується з АП за допомогою процедури асоціативного читання і переписується в БП типу FIFO. Цикл безадресного читання даних з БП значно менше циклу асоціативного читання з АП, що дає можливість прискорити обчислення.

Вільний $ОМ_j$ приймає команду з БП і приступає до її виконання. Адреса I_i на час виконання команди зберігається в РВ. Одночасно з цим скидаються в нульове стан ознаки в відповідну комірку АП і блокується запис нової інформації в зазначену комірку. Поки при-знаки знову НЕ будуть встановлені в одиницю, команда вважається неготовою і не може бути зчитана повторно іншим ОМ. Разом з цим в БУ запускається таймер T_j на час, який достатній для виконання самої найдовшої команди. Число таймерів має дорівнювати числу ОМ.

Після успішного виконання команди результат з $ОМ_j$ через КС1 записується в АП за адресою N_i. Разом з цим скидається таймер T_j і розблокується адресний запис в комірку з адресою I_i, тобто ця комірка може бути використана для формування іншої команди з відповідним ім'ям.

У разі відмови $ОМ_j$ спрацьовується таймер T_j (закінчується ліміт часу на виконання команди). $ОМ_j$ відключається від КС і команда активується шляхом установки признаків $d_2d_1a = 111$ в комірку АП з адресою I_i, який зберігався в $РВ_j$. Вільний ОМ зчитує цю команду і вона виконується по-повторних, як зазначено вище.

Оскільки підготовка алгоритму до реалізації для потокової системи виконується без урахування конкретного числа ОМ в системі, то є можливість продовження обчислення до тих пір, поки в системі буде залишатися хоча б один працездатний ОМ.

Основним недоліком системи є складність асоціативної пам'яті і велика тривалість циклу асоціативного пошуку даних, що обмежує обсяг пам'яті і знижує продуктивність системи.

Труднощі реалізації асоціативних запам’ятовуючих пристроїв великого обсягу призводить до необхідності їх емуляції із застосуванням інших технічних засобів, наприклад, спеціалізованих процесорів, що дозволяє збільшити об'єм пам'яті, але істотно знижує продуктивність, оскільки процес емуляції вимагає великих затрат часу . Крім того, при такій організації системи неможливо використовувати осередки АП після зчитування команди для інших команд, що потрібно, наприклад, при ітераційних обчисленнях і обчисленнях в конвеєрному режимі. Все це знижує ефективність паралельних обчислень.

3.Існуюча обчислювальна система з пам’яттю з довільним адресним доступом (рис 4.7).

Команда починає формуватися в комірках ПДД, два розряду яких (d і a) є ознаками, що вказують на наявність в ПДД одного операнда D_1 і актора A з часом очікування його результату виконання T. Фактично T і A можуть розглядатися як один інформаційний об'єкт. Запис зазначених об'єктів для і-ї команди здійснюється за адресою I_i. При надходженні з КС1 другого операнда D_2 для даної команди, він записується безпосередньо у відповідні розряди регістра буферної пам'яті (БП) типу FIFO, куди одночасно з ПДД переписуються A, T і D_1. При цьому комірка ПДД звільняється(чому так можна, буде описано нижче).

Таким чином, в БП формується готова для виконання команда, яка потім через КС2 передається у вільний ОМ. При цьому вміст відповідної комірки ПДД зберігається для забезпечення повторного формування команди в разі відмови ОМ. Одночасно з надходженням команди $\langle A, D_1, D_2, T \rangle$ в $ОМ_j$ вона повністю записується в регістр тимчасового зберігання $РВ_j$ на вхідній шині відповідного обчислювача. Саме через те, що записується уся команда у відповідний РВ, то немає необхідності залишати зайву інформацію в ПДД. Уся необхідна інформація для повторної операції, в разі відмови процесора, буде присутня в РВ. При успішному виконанні операції, тобто коли час її виконання не перевищило

ліміт часу виконання команди в $ОМ_j$ встановлений відповідним таймером $Т_j$, що знаходиться у відповідному тимчасовому регістрі $РВ_j$ і запускається відразу із записом в нього та $ОМ_j$ команди. Тож немає необхідності тримати зайві таймер в БУ. Результат операції надходить через КС1 в ПДД за адресою N_i.

Якщо при виконанні операції виникла відмова $ОМ_j$, тобто ліміт часу виконання операції, заданий відповідним $Т_j$, був перевищений, $ОМ_j$ вважається несправним і блокується (відключається від КС1 і КС2), а значення D_2 з $РВ_j$ знову передається в СФК. Оскільки значення А і D_1 загублені не були (відповідна комірка ПДД не була змінена), то команда повторно записується в БП, що дає можливість реалізувати наступну успішну спробу виконання команди в справному ОМ.

Розглянутий підхід дозволяє скоротити час реконфігурації і відновлення системи при відмові ОМ, що є дуже важливим фактором для систем реального часу.

4.3. Аналіз надійності обчислювальної системи

Надійність - властивість об'єкта зберігати в часі у встановлених межах значення всіх параметрів, що характеризують здатність виконувати необхідні функції в заданих режимах і умовах застосування, технічного обслуговування, ремонту, зберігання і транспортування.

Так як розроблена система є відновлюваної, то надійність її компонент характеризується інтенсивністю відмов λ і коефіцієнтом готовності Кг.

Інтенсивність відмов величина адитивна і визначається для кожного компонента як:

$$\lambda = \Sigma \lambda i * Ni,$$

де λi - інтенсивність відмов модулів, роз'ємом і пайок,

Ni - кількість відповідних компонент.

Для конкретності визначимо на яких елементах і інтегральних схемах може бути побудована розглянута система.

В якості комутатора 2 можна використовувати набір інтегральних схем К155КП1.

Для даної схеми маємо наступні характеристики:

λ0 = 1.4 * 10-7 час-1. Для системи необхідно 64 мікросхеми даного типу. Як блоків 5 і 6 регістрів даних і адреси можна використовувати інтегральну схему К555ІР23:

λ0 = 1.016 * 10-7 час-1. Для системи необхідно 4 мікросхеми даного типу. Як блоків 7, 8 пам'яті операндів і керуючих слів можна використовувати інтегральну схему К573РУ16:

λ0 = 1.35 * 10-6 час-1. Для системи необхідно по 32 мікросхеми даного типу для кожного з блоків 7 і 8, тобто всього 64.

Як блоків 3, 9 буферної пам'яті даних і буферної пам'яті команд відповідно можна використовувати інтегральну схему К564РП1 або її закордонний аналог CD400039:

λ0 = 4.445 * 10-7 час-1. При довжині буферної пам'яті в 20 слів для системи необхідно по 40 мікросхем для буферної пам'яті даних і 120 мікросхем для буферної пам'яті команд, тобто всього 160.

Як обчислювального блоку можна використовувати будь-який мікропроцесор з шиною даних не менше 64 біта. Із сучасних і відомих моделей найбільш підходить мікропроцесор Pentium корпорації Intel. Так як інформація про інтенсивність відмов даного процесора корпорацією Intel не афішується, то візьмемо максимально можливу для даного класу елементів:

λ0 = 2 * 10-5 час-1. Для нашої системи для прикладу і здійснення необхідних обчислень визначимо число обчислювальних блоків рівним 8.

Визначимо також такі складові як інтенсивність відмови пайок:

λ0 = 1.5 * 10-6 час-1 (при Ni = 750)

інтенсивність відмови друкованих провідників:

λ0 = 2 * 10-5 час-1 (при Ni = 200).

У підсумку маємо сумарну величину інтенсивності відмов:

λ = 3.483864 * 10-4 час-1 = 0.0003483864 годину-1.

Тоді напрацювання на відмову буде дорівнює:

Тно = 1 / λ = 2870.3761 годин.

А коефіцієнт готовності:

Кг = 1 / (1 + λ * Твосст) = 1 / (1 + λ * (Тз + Тп / 2)),

де Твосст = (Тз + Тп / 2) - середній час відновлення,

Тз - час заміни (0.25 години),

Тп - час перевірки одного модуля (1 година),

буде дорівнює:

Кг = 0,99973878.

Таким чином, можна зробити висновок, що метод підвищення надійності, обраний нами, дав позитивні результати.

4.4. Висновки

В даному розділі було проаналізовано результати роботи відмовостійкої обчислювальної системи, що керується потоками даних. Для аналізу та порівнянь було обрано дві вже існуючих обчислювальних системи: одна, що використовує асоціативну пам’ять, а інша використовує пам’ять з довільним адресним доступом. Усі системи було розписано та проаналізовано їх переваги та недоліки. Схеми даних розробленої та існуючих архітектур зображені на рисунках 4.5, 4.6, 4.7. Для усіх обчислювальних систем були розроблені програмні моделі(емулятори), котрі мають необхідний інтерфейс для тестування, перевірки та аналізу даних архітектур. Функціонал емулятора дозволяє обирати кількість

обчислювальних модулів(процесорів) в архітектурі, змушувати їх відмовляти(для перевірки відмовостійкості) та відновлюватися. Для моделювання роботи розроблений псевдокод, що керується файлами користувацьких налаштувань(набір команд для процесорів, час їх виконання та інше).

У якості алгоритму для аналізу був обраний симетричний блочний алгоритм шифрування IDEA. Цей алгоритм досить просто представити у вигляді графа, тож запрограмувати його на потоках даних не є проблемою. Даний алгоритм й досі використовується, тож перевірка на ньому матиме й деяке практичне значення.

У результаті аналізу було сформовано таблиці 4.1, 4.2, 4.3, 4.4 та діаграми на рисунках 4.2, 4.3, 4.4. З підсумкової таблиці 4.4 видно, що розроблена архітектура ефективніша за існуючі на 0-15%. Отже, розроблений алгоритм є не гіршим за вже існуючі, а в більшості випадків є навіть кращим. Зрозуміло, що ці значення обраховані для конкретного алгоритму і конкретних наборів команд з часом їх виконання, тому для інших результат може відрізнятися.

Також, математично було доведено надійність даної системи. Отримане значення коефіцієнту готовності 0.9997 є достатньо високим показником надійності системи.

5. БІЗНЕС-МОДЕЛЬ

5.1. Опис проблеми і дерево проблем.

5.1.1. Опис проблеми.

Проект спрямований на розробку та випробування методу, метою якого є підвищення ефективності реалізації паралельних обчислень у розподілених системах. Розробка методу включає спосіб ефективного розподілу завдань між вузлами розподіленої системи. Тестування передбачає створення програмного забезпечення для моделювання розподіленої системи для подальшого використання цього методу в системі.

Розробка методів підвищення ефективності реалізації паралельних обчислень у розподілених системах актуальна в зв'язку з тим, що обсяг інформації зростає з кожним днем, інформація обробляється, циркулює, перезаписується і зберігається. Все це робиться у великих системах, таких як центри обробки даних, щоб зберігати, або корпорації, що займаються наукою про дані, для обробки і виведення статистичних даних. Таким чином, виникла необхідність прискорити ці процеси взаємодії з даними, так з’явилися паралельні обчислювальні системи, а потім проблеми швидкодії цих систем, шляхом створення динамічних задач для системи. Так як кожна велика система має набір операцій, котрі вона може виконувати з наборами даних і деякі вхідні дані, а на виході ви отримаєте різні результати і в даний час прискорення роботи таких систем є частиною прискорення такого процесу вибору операцій для введення даних, щоб результат задовольнив вимоги, і ці операції вибираються динамічно з використанням окремо розроблених модулів, котрі, по суті є вирішенням проблеми. В даний час такі рішення приймаються графами, що динамічно доповнюють себе самі. Однак, як вже було сказано вище, методів, що забезпечують абсолютне прискорення роботи таких систем немає, тому що для цього необхідно враховувати типи систем, їх характеристики і обсяги інформації,

з якої відбувається дія. Розроблені в даний час методи використовують сплановані так звані «щоденники» проблем, з якими майбутні дії будуть обрані системою, він не є оптимальним, так як етап формування таких «щоденників» займає деякий час на початку роботи системи і, відповідно, тим більше обсяг і складність завдань, які мають більше часу, щоб витратити на нього. Існує також версія статичних задач розпаралелювання в стадії розробки, система відображає максимальний рівень роботи на оптимальній швидкості, так що це можливо, тільки якщо розробник знає і описує в коді всі можливі сценарії з використанням програмних потоків.

Основна увага проекту - відповідно до його мети, прискорювати генерацію і розподіл завдань для розподіленої системи. Прискорення повинно відбуватися за рахунок генерації наступних завдань на етапі здійснення поточного (попереднього). Використання цього методу дозволить прискорити роботу всіх маніпуляцій з інформацією і її зберіганням.

Метою проекту є винахід і тестування системи для прискорення паралельних обчислень у розподілених системах. Для цього виділимо існуючі методи для підвищення виявлення їх переваг та недоліків. По кожному з них вибрати шлях розвитку проекту виявленими перевагами і недоліками. Прискорення роботи центрів обробки даних і центрів зберігання даних та прискорення обробки інформації забезпечить проект даного винаходу спосіб додаткової оптимізації. Крім того, один з напрямків дослідження це створення тренажера для отримання статистичної інформації для демонстрації і порівняння з аналогами даного винайденого методу. Відповідно до цих результатів, необхідно виконати пошук інвестицій для установки або заміни цих систем або заміни старих і менш оптимальних. Для того, щоб залучити сторонніх розробників для подальшого випуску, необхідно запустити безкоштовний пакет

програмного забезпечення, що дозволить розробникам створювати свої власні методи і програмне забезпечення на основі методу, це буде використаний для популяризації серед розробників.

Метод заснований на заміні «щоденників» і усунення статичного розпаралелювання. Система «щоденника», буде замінена на орграф, на вузлах якого операції системи і ребрах графа послідовність виконання, тобто на початку системи ми маємо набір вхідних вузлів, так звані стартери, а потім продуктивність кожного вузла, який генерується за допомогою наступного ребра графа рухається від поточного стану і виконується в наступному вузлі з наступним поколінням.

5.1.2. Дерево проблем.

Дерево показано рисунку 5.1.

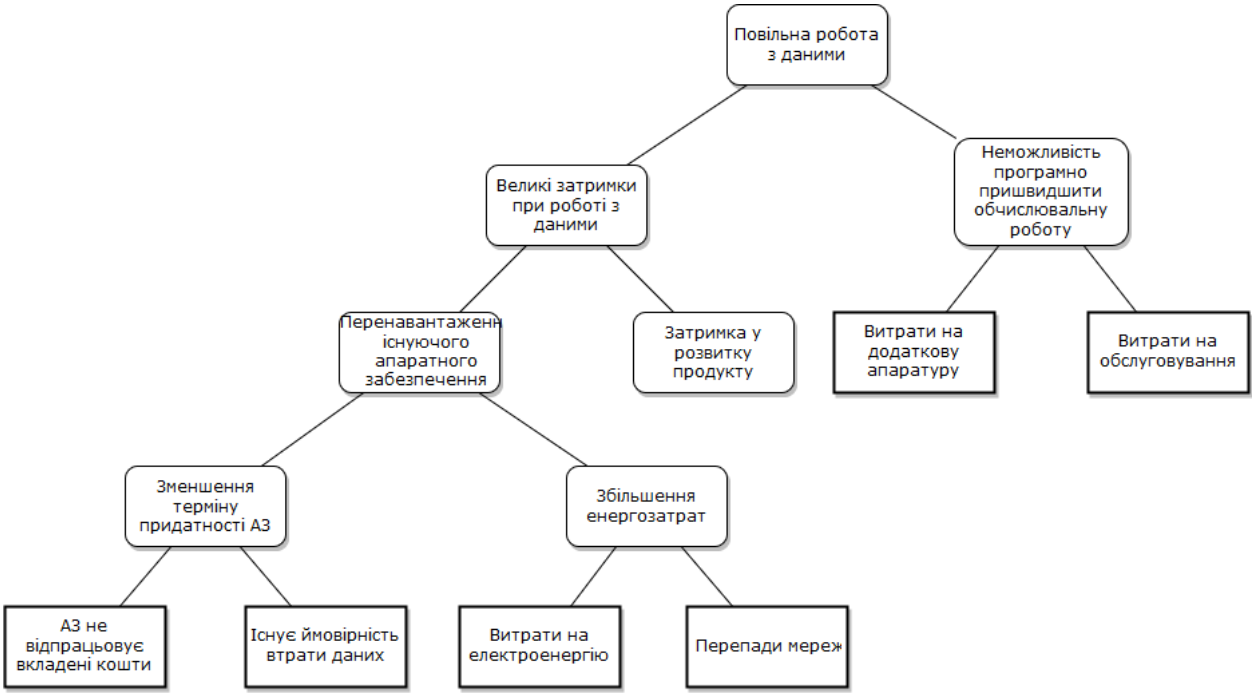


Рисунок 5.1 – Дерево проблем.

5.2. Аналіз зацікавлених сторін проекту.

5.2.1. Зацікавлені сторони проекту.

Зацікавлені сторони проекту, їх важливість і інтерес, наведені в таблиці 1.

Таблиця 5.1. Аналіз зацікавлених сторін

Група зацікавлених осіб	Інтереси групи в проекті	Умови довгостроково го співробітництв а з проектом	Важливість	Інтерес
Внутрішні зацікавлені сторони проекту				
Розробник проекту	Виконання проекту; Досягнення цільових показників проекту	Подальший розвиток проекту; Приток інвестицій	10	10
Керуючий склад проекту	Досягнення цільових показників проекту; Подальший розвиток технологій проекту; Отримання та збільшення прибутку	Подальший розвиток проекту; Приток інвестицій; Збільшення особистого доходу	8	10

Інвестори проекту	Отримання зазначених прибутків від участі у проекті; Подальший розвиток проекту	Збільшення прибутку від інвестицій; Вигідніші умови підтримки проекту	7	10
Зовнішні зацікавлені сторони проекту				
Підприємства, що займаються паралельними обчисленнями	Використання продукту проекту для прискорення роботи; Якість власного продукту; Власний технологічний розвиток	Розвиток проекту	6	8
Підприємства, що використовуют ь розподілені системи	Використання продукту проекту для збільшення якості та швидкості власного продукту;	Розвиток проекту	6	8

Наукові дослідницькі центри з великим обсягом інформації	Прискорення процесу досліджень; Прискорення процесу передачі інформації; Автоматизація	Покращення старих технологій методами внесення змін у існуючі	5	6
Побічні зацікавлені сторони проекту				
Розробники ПЗ	Збільшення ефективності використання потокових технологій;	Оновлення технологій	6	9

5.2.2. Аналіз зацікавлених сторін проекту.

Як показано в таблиці 1 розділу 5.2.1, зацікавлені сторони були розділені на 3 основні групи.

1.Внутрішні учасники проекту.

Тут ви можете знайти людей, безпосередньо зацікавлених в проекті та його подальшому розвитку і монтажні компанії про установи і т.п., в той числі інвесторів, для яких важливо, щоб отримати прибуток з проекту, розробник проекту, який сам по собі є глава всього, що відбувається, як всередині команди, так і за межами проектної команди, контроль за здійснення проекту.

Ця група має менш серйозне ставлення до проекту, але без його існування і подальшого розвитку можливо, серед них майбутні власники акцій, які включені і ті з першої групи, тому їх важливість і інтерес тісно перетинається з

учасниками першої групи, а також акціонери належать до бічної грані, інтерес, який може бути менше.

Побічна співробітників, їх значення не настільки висока, особливо на ранніх стадіях розвитку, по-перше, число невелике, а по-друге, вони не носять настільки сильне значення, оскільки безпосередньо не пов'язані з запуском або застосування розробленого методу для використання однак, якщо проект отримає подальший розвиток, їх число буде збільшуватися, і співробітники, які були присутні на старті отримують великі доходи.

2. Зовнішні зацікавлені сторони.

Це все, хто зацікавлений, щоб мати доступ до використання методу, тобто «цільової аудиторії» методу, він включає в себе всі компанії, підприємства, компанії, установи, що працюють з інформацією, в тій чи іншій формі, а також можуть бути носіями статистичних даних, або метод тестової платформи.

3. Побічні зацікавлені сторони.

Протиріччя в тому, що ця група має низький інтерес і низьке значення, є потужним двигуном для просування методів, після того, як незалежні програмісти можуть використовувати метод, щоб створити свої власні продукти в майбутньому, що буде приносити постійне користування.

5.3. Опис дослідницького проекту і технології.

Дослідження продукту в рамках наукової роботи є метод підвищення ефективності реалізацій паралельних обчислень у розподілених системах, який є науковим назвою продукту.

Цей метод призначений для використання будь-якої установи, які працюють з інформацією, що зберігають його. Цей метод дозволить спростити процес роботи установа з інформацією, використовуючи технологію орієнтованого ациклічного орграфа (DAG). В якому дуга - це напрямок, шлях, і

вузли операція, які повинні бути завершені на даному етапі і генерують наступну операція, тим самим усуваючи необхідність завчасного планування дій або статичного програмування на етапі написання програмного коду з система.

Ациклічний орієнтований граф - варіант орієнтованого графа, в якому немає орієнтованих циклів, тобто шляхів, Починається і закінчується в одній і тій же вершині. Спрямований ациклічний граф є узагальненням дерева.

На ранніх етапах розвитку цього методу тестування буде проводитися по збору статистичних даних про продуктивність, імітатор центру обробки даних з поточної обробкою даних, такими як перетворення або кодування, імітатор буде створено за допомогою його власне написання коду. На наступних стадіях способу застосування на підприємствах і т.д., дані підприємства будуть служити в якості так званої платформи для тестування і збору інформації для подальшого вдосконалення методу, тому метод розвитку ніколи не зупиняється.

Для реалізації способу, безліч етапів, виділених на абстрактному рівні:

1. Аналіз різних підходів до прискорення виділення динамічних систем і їх переваг і недоліків;
2. Створення систем, заснованих на орграфі;
3. Контрольна система в порівнянні з існуючими методами;
4. Написання тестової платформи (ЦОД імітатора)
5. Збір статистичних даних і порівняння їх з реальною статистикою;
6. Усунення дефектів, якщо це можливо, на основі даних, отриманих з тестової платформи.

5.4. Бізнес-рішення та основні характеристики бізнес-продукту.

5.4.1. Огляд продукту

Результати науково - дослідної роботи буде програмне забезпечення і документацію до нього, програмне забезпечення допоможе прискорити роботу

алгоритмів, або прискорити обмін інформацією. Далі буде розглянуто принцип розробки, використання продукту, способи підтримки і подальшого розвитку, що дозволить абстрактне уявлення про продукт, який отриманий в результаті дослідницької роботи. Як буде обговорюватися нові можливості і способи введення в виробництво різних видів програм, а також можливість отримання програмного забезпечення і подальше використання в «будинку», як буде описана в необхідних вимогах для установки програмного забезпечення.

5.4.2. Опис продукту

Результат дослідницької роботи буде, проте слід зазначити, що програмне забезпечення не є основним продуктом виробництва, засноване на програмному забезпеченні лежить метод розподілу операцій у розподілених системах, тобто результат буде точно таким методом, який може бути встановлений або інтегрованим в різних виробничих системах, що працюють з великими обсягами інформації.

Основна перевага способу використання продукту є швидкість обробки інформації і колекції різних програмних продуктів, може знадобитися такий спосіб: в центрах обробки даних з великим оборотом інформації, автопідприємства, надаючи розробникам.

Дата - центри зможуть прискорити обробку даних і подальшого їх збереження.

Підприємства з додатковою інформацією поводженням з використанням цих технологій будуть у багато разів швидше, ніж сортування зберігати і перенаправляти інформацію.

Розробники програмного забезпечення, використовуючи цей метод буде мати можливість писати багатопоточні програми без відстеження та координування потоків і зручне випробування.

Головною особливістю конструкції є простота інтеграції, тобто, цей спосіб можна легко реалізувати у вигляді додатку або бібліотеки підключений до сторони проекту або інтегровані в процес, що дозволяє розподілити поточний метод розрахунку навантаження.

Процес методу «продажу» буде метод інтегрування методу в свій власний проект або повністю на виробництві або компанії, і її подальша підтримка для оплати допоміжних послуг, щомісяця і для використання технології, в випадок користувальницької бібліотеки буде забезпечено їх власними зменшених розробниками проекту тільки інтеграція, але не підходять для великомасштабних проектів. Для того, щоб отримати доступ до технології замовляються інтеграції в проект, і обов'язково подальшої підтримки, який буде прикріплений окремий фахівець.

Продукт знаходиться під безперервним розвитком, що дозволить постійний спосіб додавання і адаптації його до різних типів даних, це і є одним із завдань оптимізації розподілених систем, тому що різні типи інформації і різних систем, обробки і зберігання різних бізнес-рішенням є постійною підтримкою і постійний розвиток методому, так як програмне забезпечення буде встановлено точно в момент фактичної версії методу, тому для оновлення та додаткової підтримки необхідне фінансування команди підтримки безпосередньо для отримання нової версії продукту. Так як результат складання є негайним прискоренням статистичних даних для кожного власника системи, система обробки інформації буде використовувати свій власний метод зберігання і прискорення,

Надалі цей метод буде рухатися до сервера інтеграції на більшу частину, що дозволить значно прискорити обмін інформацією, але основний розвиток має стати платформою для запуску додатків, які будуть працювати швидше за допомогою цього методу, а також випуск повної бібліотеки для розробників.

5.4.3. Конкурентні переваги рішення.

Серед основних критеріїв конкурентоспроможності є:

1. Новизна проекту;
2. Конкурентоспроможне здатність персоналу;
3. Передові технології;
4. Передові технологічні процеси і обладнання;
5. Науковий рівень розвитку;
6. Науковий рівень системи.

Технологічні переваги методу (проект):

1. Відсутність так званого «блог» програмного забезпечення паралельного програмного забезпечення. «Блог» програмне забезпечення являє собою програмний модуль, який почне працювати прямо з програми для роботи з даними і програмами послідовності операцій програми в своїх відповідних потоках в процесі виконання коду програми, «Щоденник» динамічно забезпечує дані програми, які можуть виконуватися паралельно, і повинні чекати попередніх кроків. Метод для досягнення динамічного розпаралелювання, що дозволяє позбутися від «щоденника» і часу, відведеного для його роботи і часу відносяться до нього за допомогою програми.
2. Позбавлення від проблеми статичного розпаралелювання, тобто програміст не описує безпосередньо в потоці даних програмного коду, що спрощує роботу розробника в майбутньому.
3. Реалізація динамічної бібліотеки з тестовим пакетом для забезпечення здійснення способу виробництва перевірити результати методу на реальних даних.
4. Зручний пакет розробника забезпечить додаткову мотивацію для використання методи «вільної» сторони і розробників.

5. Крос-платформна, цей метод, а саме, його реалізація не вимагає підключення до Інтернету, тобто, працює без підключення до мережі, навіть місцеві.
6. Відповідно, в рамках пункту 5, програмна реалізація методу має підвищену стійкість у разі переривання інтернет-мережі вимикаються тільки по-функції програмного забезпечення, а також метод розпаралелювання продовжує працювати.
7. Система моніторингу в режимі реального часу.

Основні переваги проекту:

1. Оскільки проект в майбутній команді (фірми, компанії і т.д.) не використовує кредитний підхід, отже, може дозволити зручний і постійний формат цінової політики;
2. Весь персонал, який буде займатися технічним обслуговуванням і модернізацією систем і устаткування від клієнтів, постійні курси навчання і розвитку, участі в конференціях;
3. Оптимізація потужності, так як метод системи прискорення постійно розвивається і перебуває в постійному процесі збору статистичних даних, це дозволяє оновлювати програмне забезпечення і апаратні засоби в найкоротші терміни;
4. Швидка динаміка в оновленнях, а також підтримка, відповідно до нових статистичними даними, зібраними в тому числі інших установ, відповідно, тим більше клієнтів, тим вище швидкість оновлення і доповнення методів.
5. Відсутність низькокваліфікованих працівників;
6. Установка і оновлення системи прискорення відбувається безпосередньо, без посередників;
7. Інтернет метод тестування з прикладами.

Слід також звернути особливу увагу на пункти тестування, тому що впровадження таких методів виробництва, клієнт хоче, щоб переконатися, що ця система дасть бажані результати, це завдання вирішується шляхом реалізації окремого повноправним, незалежний тест користувач бібліотеки, яка на додаток до системи моделювання роботи на реальних даних і моніторингу, для створення системи для поліпшення підприємства, а саме програмне забезпечення, на якому він встановлений. Цей аспект має істотну перевагу, оскільки до способу установки може бути повністю протестований, і результати аналізували, щоб поліпшити клієнт.

Відсутність «Щоденник» (пункт 1 із технологічних переваг) забезпечує можливість реалізації динамічного ациклічний орграф, так званий файл directed acyclic graph, приклад графіка, показаного на рисунку 5.2, він має перевагу в порівнянні з іншими реалізаціями, які встановлюють на роботу Кращі зберігаються окремо, а самі процеси виконують окремо, тобто дуга графа, напрямок програми, а вузли є операціями, які накопичуються в загальних операції басейну.

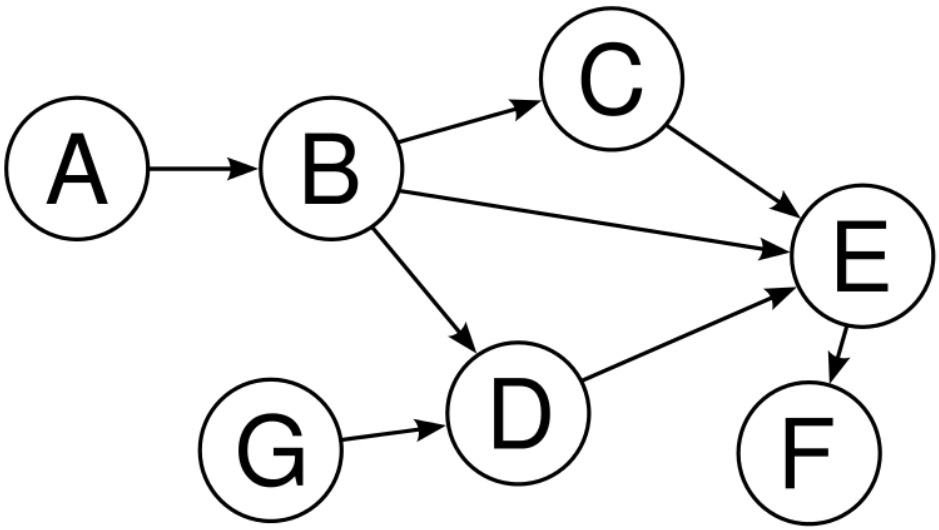


Рисунок 5.2 – Directed acyclic graph.

Основні особливості паралельної моделі програмування, тим вища продуктивність програм, використання спеціальних методів програмування і, як наслідок, тим вище складність проблем програмування програми передачі. Паралельна модель не має унікальні властивості. У паралельних моделях програмування мають проблеми, незвичайні для програміста, який звик займатися в послідовному програмуванні. Серед них: операції управління з безлічі процесорів, організації міжпроцесорної передачі даних і т.д. D. Чотири основні переваги паралельних програм можуть бути сформульовані:

1. паралелізм;
2. універсальність;
3. місцезнаходження;
4. модульність.

5.4.4. Клієнти і споживчий ринок

Як уже згадувалося в розділі 5.2, аналіз зацікавлених сторін, основними замовниками і споживачами методу є підприємства і установи з великим оборотом даних. Детальний збір інформації про клієнтів, зібрані в таблиці 5.2.

Таблиця 5.2. – Клієнти

Тип закладу	Опис установи, проблема якої повинна бути вирішена за допомогою методу	Споживчий рейтинг
Центри обробки даних	Такі місця кожен другий обробляти великі обсяги інформації, які в цьому випадку повинні бути збережені в процесі, інформація стає з кожним	10

	днем все більше і більше, і способи обробки, зберігання та передачі занадто повільно, що призводить до втрати швидкості і ефективність інституту і втрати точності. Спосіб підвищення ефективності паралельних обчислень у розподілених системах частково вирішує цю проблему, тому що це його сутність, прискорити роботу з даними з використанням методи даних, що циркулюють в системі буде набагато швидше проходить обробку відповідно більшу кількість за одиницю часу, що дозволить підвищити ефективність даних - центрів і підвищити їх потенціал.	
Центри зберігання даних	Як правило, ці компанії діють статистичні дані і великі бази даних, такі як бази даних з ціновою політки, або списки співробітників, або наборів товарів і т.д., для основи повинна обробляти запити наборів, краще працювати асинхронно в окремому потоці, як правило, такі підприємства мають стабільне	8

	обладнання на вузлах, що дозволяє легко на його основі, щоб встановити програмне забезпечення, яке реалізує метод підвищення ефективності запитів до бази даних, такий підхід дозволить позбутися черг запит в, або перенавантажених об’єднаних запитів, підвищити ефективність роботи підприємства, а також програмна реалізація методу дозволяє зручно контролювати і збирати статистичні дані в окремій базі даних, результати застосування.	
Державні інформаційні установи	Як правило, ці установи мають широке поширення інформації, що робить процес обробки і зберігання є тривалим і громіздким, а також важливий в таких установах є момент, щоб фільтрувати дані та інформацію про конфігурацію, фільтри пошуку і відображення для конкретних тегів, звичайно є динаміка в процесі інформаційного потоку, пропонований спосіб динамічного розподілу завдань для розподілених систем значно	6

	прискорить процес пошуку і фільтрації даних шляхом установки структур ациклічних орієнтованих графів розвантажити систему, тому що такі установи, як правило, обладнанні застаріло і можуть не витримати навантаження.	

Як видно з таблиці, яка дається великими клієнтами є великими компаніями, що працюють з потоком інформації, перевага цього інвестиції мити і прив’язаність для кожного великого клієнта, але недолік є великим ризиком і складністю в пошуку нових клієнтів. Оцінка споживачів(таблиця 2) вказує на загальний коефіцієнт необхідності для конкретного типу установ щодо того, чи є вони готові до реалізації подібних технологій, і якщо вони роблять, вони необхідні, наприклад, для держави. Установи подібний метод введення було б корисно, в зв'язку з прискоренням роботи і розвантаження персоналу, але готовність встановити і оновлювати таке програмне забезпечення менше, ніж на приватних підприємствах.

Як ми знаємо, сегментація ринку має 3 рівня:

1. Стратегічна сегментація він забезпечує набравши продуктів на ринок і опис загальних характеристик товарів, представлених на ринку, проект заснований на розробці методу, тобто наукова новизна.
2. Сегментування товарного ринку ґрунтується на потребах потенційних покупців, в даному випадку, обробка різних типів і прискорення систем.

3. Конкурентна сегментація, яка встановлює принцип конкуренції в даному сегменті ринку, основою цього методу є швидкістю, тобто швидкості є вирішальним фактором в конкурентній боротьбі.

5.5. Унікальна цінність пропозиції (продукту дослідження).

Як уже згадувалося раніше ці рішення, є безліч, але всі вони, або статичні, або використовувати так званий метод планування «щоденник» заснований на децентралізованої ациклічний орієнтований граф, який є так звана новизна рішення, не витрачатиме час на планування процесу.

Оптимізація продуктивності може бути досягнуто на різних рівнях за допомогою цих графіків, розглянемо деякі з них:

1. рівень процедури - це рівні різних секцій однієї і тієї ж програми повинні здійснюватися паралельно. Ці ділянки називаються процесами і є послідовностями процедуру. Проблеми діляться на, по суті, незалежні частини таким чином, що менш можливо виконати обмін даними між процесами, які вимагають порівняно великої кількості часу. У різних додатках, то ясно, що цей рівень паралелізму в будь-якому випадку не обмежується розпаралелювання послідовних програм. Існує велика кількість проблем, які потребують паралельних структур типу, навіть якщо такий же, як на рівні програмного забезпечення, користувач має тільки один процесор. Застосування (основне) - загальна паралельна обробка інформації, де це може бути застосована проблема поділу повинна бути вирішена в паралельних завданнях - компоненти, які вирішуються багатьма процесорами, щоб збільшити продуктивність обчислень

2. рівень арифметичні вирази - арифметичні вирази виконуються паралельно компонентами, з набагато більш простими синхронними методами. Якщо, наприклад, він являє собою добірку арифметичних виразів масивів, це дуже легко синхронізувати розпаралелення, тому що кожен процесор піддається елемент матриці. При застосуванні елементів п обробки п * може отримати суму двох матриць порядку N * N для виконання однієї з операцій додавання (інший, ніж час, необхідний для читання і запису даних). Це невід'ємний рівень векторизації і так званих даних паралелізму. Останнє поняття відноситься до деталей паралелізації, а саме - з його поширенням на оброблених даних. Майже кожен елемент даних тут підкоряється своїм власним процесором, так що дані про те, що машина фон Неймана була пасивною.
3. Біти рівня - це відбувається на рівні розрядного паралельного виконання операцій в межах одного слова. Паралелізм на рівні бітів можна знайти в будь-який запущеної процесорі. Наприклад, 8-бітове арифметико-логічний пристрій бітової паралельної обробки виконується на апаратному рівні.
4. Паралелізм на рівні потоків - Одним із шляхів вирішення цієї проблеми пов'язана з реалізацією концепції паралелізму на рівні потоків (паралелізм завдань- TLP). При роботі на роботу не в змозі завантажити всі функціональні можливості роботи пристрою, ви можете дозволити процесору виконувати більше однієї задачі, тим самим створюючи другий потік, тому він завантажений, що стояли без праці пристрій. Існує аналогія, це багатозадачна операційна система(ОС): процесор не простоював, коли завдання знаходиться в стані очікування (наприклад, завершення введення-виведення),

перемикачі ОС для виконання інших завдань. Найбільш ефективна сьогодні стали архітектура з одночасним виконанням потоків (Simultaneous Multi-Threading). У цій ситуації, при кожному новому циклі виконання в будь-якому приводі може бути відправлена будь команда потоку - в залежності від обчислювального алгоритму, розпаралелювання на рівні потоку TLP, на відміну від ILP, знаходиться під контролем програмного забезпечення. Віртуальна Нить створюється шляхом виділення із фізичних процесорів, два або більше логічних процесорів. Класичний приклад такого підходу є технологія Hyper Threading(HT) від Intel. У зв'язку з тим, що під час ходу, як правило, не всі модулі виконавчих процесорів беруть участь, вони можуть бути завантажені проблеми паралельного потоку. Зрозуміло, що половина продуктивність не збільшувалася, як паралельні потоки використовують спільну пам'ять кеш-пам'ять і т.д. (З тих же втрати виникають через синхронізації та інструкції паралелізації), але вона зазвичай зростає на 35-50%. Недолік технології TLP є виникненням конфліктів, коли одна нитки потребує результатами іншого договору, що призводить до очікування і зростання числа циклів, необхідних для виконання інструкцій.

5.6. Доходи і витрати.

5.6.1. Витрати.

Для того, щоб прийняти рішення щодо інвестиційного проекту всі витрати, пов'язані з його реалізацією, слід розділити на інвестиції та виробництво.

Загальна вартість проекту включає в себе:

1. Вартість формування основного капіталу включає початкові і постійні інвестиції;
2. Витрати на формування оборотного капіталу;
3. Виробничі витрати.

Всі інвестиційні потреби підприємства можна розділити на три групи:

1. Прямі інвестиції - безпосередньо необхідні для реалізації інвестиційного проекту;
2. Відповідні інвестиції - інвестиції в об'єкти, безпосередньо технологічно пов'язаних із забезпеченням нормальної експлуатації;
3. Інвестиції виконують науково-дослідну роботу.

Склад первинних інвестицій є:

1. Витрати на передінвестиційних досліджень, проведення наукових досліджень і дослідно-конструкторських робіт по розробці проектних матеріалів по детальній розробці проекту і палітурки;
2. Витрати на покупку та оренду приміщень, включаючи вартість навчання в цілях розвитку;
3. Витрати на придбання і постачання машин, обладнання, інструментів і обладнання, в тому числі імпорту;
4. Витрати на приймально-здавальні випробування;
5. Витрати на введення в експлуатацію, комплексне освоєння проектної потужності і домогтися розробок технічних та економічних показників;
6. Витрати на придбання патентів, ліцензій, ноу-хау, технології та інші амортизації нематеріальних активів;
7. Витрати на навчання вводиться в дію;
8. Одноразові виплати, в тому числі гарантій і страхових компаній;

9. Збитки, що виникають в результаті створення і реєстрації підприємства (юридичні послуги з підготовки установчих документів, реєстраційних витрат компанії і реєстрації прав власності;
10. Витрати на підготовчі дослідження не включені в кошторисної вартості об'єкта;
11. Витрати, пов'язані з діяльністю персоналу під час попереднього виробництва (заробітна плата, витрати, технічне обслуговування приміщень, транспортних засобів, комп'ютерів та іншого обладнання), які не були включені в кошторисної вартості проекту.

Висновок про витрати:

Вартість проекту = вартість проектування + вартість розробки + вартість підтримка і поліпшення після впровадження + плати за компоненти та послуги + плати за рекламу і просування. На рисунку 5.3 показана схема витрат.

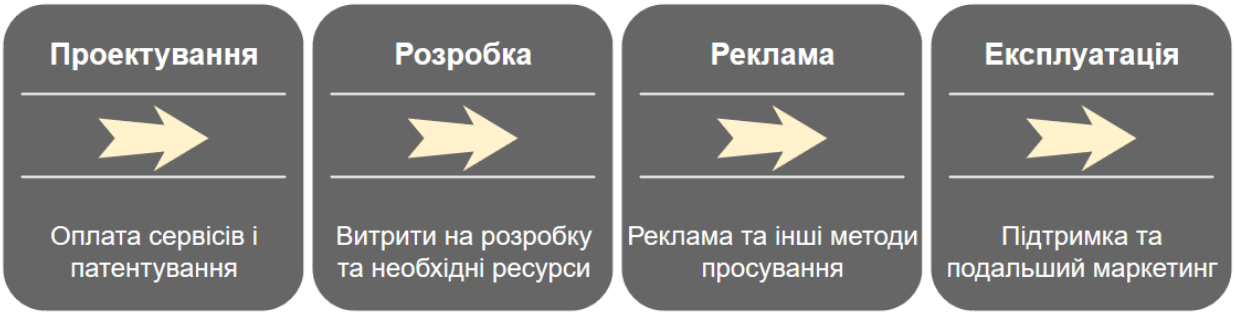


Рисунок 5.3 – Види витрат за проектом на різних етапах.

5.6.2. Доходи.

Як ми знаємо з попередніх пунктів, і на споживчому ринку, товар є способом підвищення ефективності реалізації паралельних обчислень у

розподілених системах. Основним джерелом доходу буде засобом за допомогою цього методу. Загальна рентабельність проекту в майбутньому буде розглядатися як показники рентабельності, такі як:

1. Замовлення**;**
2. Виручка;
3. Загальна сума контракту;
4. Life Time Value (загальне значення)
5. Доходи майбутніх періодів.

Замовлення -оцінка вартості контракту між компанією і клієнтом. Цей показник відображає зобов'язання клієнта платити компаніям гроші, зазначені в договорі. З моменту заснування функціонування і реалізації способу отримання контракту цієї метрики, грає важливу роль при розрахунку прибутковості проекту, загальна вартість контракту в одній установі.

Ми можемо говорити, коли послуга була надана або буде надаватися на регулярній основі протягом зазначеного в прибутку за контрактом періоду підписки (доходи). Існує фактор оригінального першого показника розраховуються по закінченню контракту, тобто, в разі успіху.

Первісна вартість контракту вважається, якщо договір визначає елементи для підтримки, доповнення або передачі вартості чого-небудь, наприклад, як виплата подальшої підтримки виробництва. TCV в кінцевому підсумку може збільшуватися або зменшуватися. Переконайтеся, що TCV також враховує одноразові витрати, оплати спеціальних послуг і періодичних платежів.

Lifetime значення - поточна оцінка майбутніх чистих доходів від клієнта протягом всього періоду його відносин з компанією. Це допомагає визначити довгострокову цінність клієнта, а також чистий прибуток на одного клієнта на підставі його залучення витрат (CAC). Щомісячна маржа на одного клієнта =

дохід від клієнта за вирахуванням змінних витрат, пов'язаних з клієнтом. Змінні витрати включають в себе всі адміністративні та експлуатаційні витрати, пов'язані з обслуговуванням клієнтів.

UDR - це гроші, які ви збираєтеся з продуктом (послуги) впорядкованості, тобто його виробництва (реалізації). Компанії визнають виручку тільки протягом терміну служби - навіть якщо клієнт платить за більшість послуг в угоді. Тому, в більшості випадків гроші враховуються в балансі в рядку так званого відкладеного доходу.

5.6.3. Таблиця витрат/доходів.

Таблиця 5.3 описує стадії розробки проекту, розділів витрат і загального доходу. Слід взяти до уваги, що перші три етапи є одноразовими, четвертий крок розраховується щомісяця, після перших етапів.

Таблиця 5.3. – Стадії розробки проекту

етап проекту	витрата	дохід	витрати	дохід	прибуток
Проектування	Оплата сервісів і патентування		500 USD		-500 USD
Розробка	Витрати на розробки та необхідні ресурси		1000 USD		-1000 USD
Реклама	Витрати на рекламу та інші методи просування		1500 USD		-1500 USD

Експлуатація	Підтримка та подальший маркетинг	Замовлення продукту, оплата за його встановлення та налаштування, оплата подальшої підтримки та оновлень	500 USD	5000 USD	+4500 USD

5.7. Бізнес модель.

На рисунку 5.4 показана загальна схема бізнес-моделі.

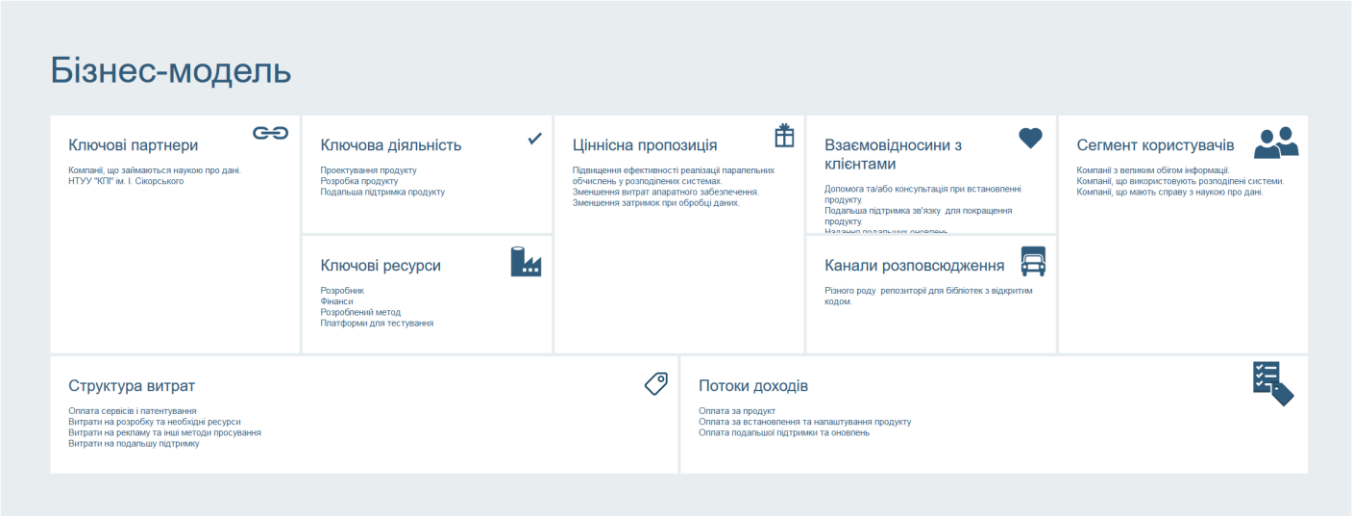


Рисунок 5.4 – Загальна бізнес-модель.

Сегмент користувачів:

Для кого? -Підприємства з великим оборотом інформації і даних.

Найбільш важливі клієнти? -Великі установи і корпорації з укладенням контракту на постійну підтримку та оновлення.

Ціннісна пропозиція:

Яка цінність методу, клієнти отримують? - Позбавлення планувальником, який надасть можливість швидкої обробки даних, що зберігаються надасть прибутку і спростили навантаження на устаткування.

Яких людей ми вирішити цю проблему? -затримки обробки даних, скорочуючи витрати на процес оптимізації роботи з інформацією.

Те, що ми задовольнити потреби користувачів? -Користувачам необхідно оновити, зберігати і обробляти інформацію швидше, ніж поточні швидкості передачі даних, оптимізація методи дозволяють.

Що відрізняє ми пропонуємо продукти? - метод прискорення, платформа для тестування і збору статистичних даних, а також підтримка оновлень програмного забезпечення та відкритим вихідним кодом.

Канали розповсюдження:

Через які канали ми хочемо досягти кожного сегмента наших споживачів? -реклама та створення вільної бібліотеки, поширити його в усіх можливих пакетів для різних мов програмування, таких як NuGet або MavenGet.

З каналів розподілу є найбільш привабливим? - можливість поширення методу, за допомогою вільних бібліотек програмного коду.

Потоки доходів:

Що користувачі готові платити? - для використання методи для прискорення роботи з інформаційними системами дозволяє поліпшити державну систему статистичної інформації і моніторингу, а також для ремонту і технічного обслуговування.

Як вони можуть заплатити? - як основними клієнтами є установи, жоден цільових клієнтів, велика частина оплати буде за контрактами, які будуть описані в термінах підтримки, тестування та оновлення контракту мають можливість продовжити або укласти, як підписка до статичного часу.

Альтернативні способи? - доходи також можуть бути оплачені пакетами для індивідуальних розробників, надаючи можливість змінювати і поліпшувати місцеві проекти.

Ключові ресурси:

Необхідні ресурси? - в тому числі необхідних ресурсів, платформ для тестування і статистичних установ обробки даних. Наукові досягнення та патенти на такі методи і фінансові ресурси.

Шляхи знайти необхідні ресурси? - безкоштовні тестові пакети для установ, які отримують вигоду від проекту і установи.

Відносини з клієнтами? - клієнт підписання типу договору, отримує набір послуг, в доповненні до впровадження технології у виробництві, це може бути сервіс оновлення, установка сучасних систем моніторингу і попередження.

Потоки доходів?- як уже неодноразово зазначалося вище, основний дохід йдуть від агентств, що працюють з інформацією, і невелика частина користувацького додатки на основі бібліотеки. Установи скласти повідомлення у вигляді щоденних контракту на кілька пакетів послуг різні і містять різні набори послуг в доповненні до безпосередньої реалізації методи.

5.8. Висновки

У даному розділі було розглянуто можливості впровадження продукту дослідження та сформовано бізнес-модель. Виявлено, що дана програмна модель

є дійсно актуальною та має можливості для реалізації. Проаналізовано проблеми, котрі даний продукт вирішує. Складено списки зацікавлених осіб, в тому числі серед тих, хто потенційно може стати спонсором проекту та/або його користувачем. З проаналізованих даних прораховано можливі витрати на реалізацію та підтримку продукту та можливі варіанти отримання прибутку.

Складена бізнес-модель показує хто стає ключовими партнерами та аналіз сегменту користувачів. Також описано необхідні ресурси та дії для успішної реалізації продукту. Показано що власне буде цікавим та приваблим для потенційних покупців та яким чином можна розпочати розповсюдження програмної моделі. Для довгосрокової підтримки описано необхідні дії для взаємодії з клієнтами. Проаналізовано на що необхідно витрати гроші та з чого можна буде отримати дохід.

6. ВИСНОВКИ

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, наведено відомості про апробацію результатів. Також надано інформацію про об’єкт, предмет та мету дослідження. Сформовано завдання, котрі були вирішені по ходу дослідження, а саме:

–проаналізовано та досліджено існуючі методи і засоби забезпечення відмовостійкості систем, виявлено їх недоліки та вибрано метод часових інтервалів, як найбільш доцільний варіант для СПД;

–проаналізовано методи динамічної реконфігурації систем;

–розроблено архітектуру відмовостійкої СПД з використанням асоціативної пам’яті;

–розроблено програмну модель для емуляції роботи архітектури;

–експериментально досліджено ефективність запропонованої архітектури за допомогою емулятора.

Описано наукову новизну та практичну цінність проекту.

У першому розділі надано загальні теоретичні відомості про паралельні обчислення, системи, що керуються потоками даних, відмовостійкість, динамічну реконфігурацію. Також розглянуто і проаналізовано існуючі рішення як архітектурні, так і програмні, описані їх переваги та недоліки. У якості архітектурних прикладів рішень було описано існуючий варіант архітектури з використанням асоціативної пам’яті та варіант з пам’яттю з довільним доступом. У якості існуючих програмних моделей було показано сучасну корпоративну громіздку систему Xilinx ISE та безкоштовну систему Icarus Verilog, котра навіть не має власного інтерфейсу користувача.

У другому розділі запропоновано та описано новий варіант архітектури відмовостійкої СПД. Дана архітектура особлива новим підходом до забезпечення

відмовостійкості. Цей підхід є модифікацією методу часових інтервалів, що використовується в описаних існуючих архітектурах. Модифікація полягає в тому, що для запуску таймерів використовується не найбільше значення часу виконання серед існуючих команд, а власний час виконання для кожного з акторів. Таким чином, навіть теоретично, це мало б пришвидшити процес діагностування несправності, що і було доведено практично у четвертому розділі. В архітектурі реалізовано динамічну реконфігурацію, адже система може працювати незалежно від того, скільки справних обчислювальних модулів. Головне, щоб був хоча б один. Введення пріоритетного буферу пам'яті команд дозволяє уникнути проблемної ситуації глухого кута. Також надано відомості про програмі засоби, що були використані для написання програмного емулятора.

У третьому розділі описано програмну модель, що реалізує розроблену архітектуру та псевдокод, котрий він оброблятиме. Для псевдокоду було описано його лексику, синтаксичні правила та особливості розширення за допомогою конфігураційного файлу. Описано структуру програмної моделі, а саме її двох основних частин: транслятору та емуляційної частини. Транслятор необхідний для переходу від більш зрозумілого користувачам псевдокоду до зрозумілого емулятору графу потоку даних. Описано три складові частина транслятора: лексичного, синтаксичного розборів та генератору графу потоку даних. Надано прикладу коду даної частини. Описано усі основні частини емулятору, їх взаємодії та відповідність того, що зображено на екрані та розроблено програмно з тим, що розроблено в архітектурі, описаній в другому розділі.

У четвертому розділі описано результати експериментів та порівнянь між розробленим та існуючими алгоритмами. Описано обраний алгоритм для тестування та перевірки ефективності розробленої системи. Це симетричний блочний алгоритм шифрування IDEA, що дуже гарно лягає на граф потоку даних,

що є основою для систем, що керуються потоками даних, отже реалізація даного алгоритму на розробленій псевдомові не стала серйозною проблемою. Описано системи, які взяли участь у порівнянні з розробленою під час дослідження. Надано експериментально отримані значення для усіх систем(для існуючих були створені аналоги розробленої програмної моделі для чистоти експерименту) під час різних варіантів стану. Протестовано варіанти зі справною роботою модулів, з відмовою деякої кількості модулів та з подальшим відновленням їх роботи. Таким чином було протестовано відмовостійкість системи. Зменшуючи кількість справних обчислювальних модулів до одного було продемонстровано роботу динамічної реконфігурації. Ситуація глухого кута оброблялась за допомогою пріоритетного буфера пам’яті команд. Окрім таблиць з чисельними значеннями експериментів було надано зручні графічні діаграми для більш наочного демонстрування результатів аналізу. З підсумкової таблиці видно пришвидшення на 0-15% порівняно з існуючими архітектурами. Зрозуміло, що збільшення ефективності залежить від конкретного алгоритму та різниці часу виконання найбільшої команди від середнього значення. Окрім цього в розділі було математично доведено надійність даної системи, адже значення коефіцієнта готовності 0.9997 є достатньо високим значенням.

У п’ятому розділі наведено опис проблем, що вирішує даний продукт, на даній основі побудовано дерево проблем. Складено, описано та проаналізовано таблиці з зацікавленими особами. Описано доцільність реалізованої програмної моделі. Складено прогноз на можливі витрати, доходи від продукту та отримання прибутку у майбутньому. На основі вище описаних даних було сформовано бізнес-модель з усіма ключовими даними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dennis J.B. Dataflow Supercomputers / J.B. Dennis // Computer. – 1980. – P. 48–56.
2. Hogenauer E.B. DDSP - a data flow computer for signal processing / E.B.Hogenauer, R.F.Newbold, Y.T.Inn // Proc. Int. Conf. Parall. Process. Ohio, August 1982. N.Y.: IEEE, 1982. - P. 126-133.
3. Watson R. A practical data flow computer / R.Watson, J.Gward // Computer,1982. – Vol.15, N 2. – P. 51-57.
4. Майерс Г. Архитектура современных ЭВМ: В 2-кн. Кн. 1. Пер. с англ / Г.Майерс. – М.: Мир, 1985. – 364 с.
5. Авиженис А. Отказоустойчивость – свойство, обеспечивающее постоянную работоспособность цифровых систем / А.Авиженис // ТИИЭР (пер. с англ.). – 1978. – Т. 66, №10. – С. 5-25.
6. Миногин А.В. Отказоустойчивые вычислительные системы / А.В. // LAP Lambert Academic Publishing. – 2011. – P. 16-24.
7. Лонгботтом Р. Надежность вычислительных систем / Р.Лонгботтом. – М.: Энергоатомиздат, 1985. – 288 с.
8. Клименко И.А. Обеспечение отказоустойчивости потоковых систем на однотипных вычислительных модулях / И.А.Клименко, В.В.Жабина // Вісник НТУУ "КПІ". Інформатика, управління та обчислювальна техніка: Зб. наук. праць. – К.: Век+. – 2010. - № 51. – С. 166-171.
9. Cassell J. - A Forgettable Year for Memory Chip Makers: iSuppli releases preliminary 2008 semiconductor rankings / Cassell J. iSuppli // Proc. Int. Conf. iSuppli, January 2009. N.Y.: IEEE, 2009. - P. 12-13.
10. Johnson D. Data flow machines thereaten the program counter//Electronic Design. 1980. N. 22. P. 255-258
11. Suzuki T., Kurihara H., Moto-oko T. Procedure level data flow processing on dynamic structure multiprocessors//J. of Inform. Proc. N.Y.: IEEE, 1982. Vol. 1. N. 5. P. 11-16
12. Takahachi N., Amamiga M. A data flow processor array system design and analysis//Proc. 10th Annual Int. Symp. Comput. Archit. Stockholm, September, 1983. P. 236-242
13. Kishi M., Yasuhara H., Kavamura Y. DDDP: a distributed data driven processor//Proc. 10th Annual Int. Symp. Comput. Archit. Stockholm, September, 1983. P. 236-242
14. Dipak Ghosal, Laxmi N. Bhuyan. Performance evaluation of a dataflow architecture // IEEE Trans Computers. 1990. V. C-39. №5. P. 615-627.
15. Gurd J.R., Watson I., Kirkham C.C. The Manchester prototype dataflow computer // Commun. ACM. 1985. V. 28. P. 34-52
16. Інтернет Ресурс Mono,URL: https://www.mono-project.com/.

17.Інтернет Ресурс MSDN,URL: https://msdn.microsoft.com/en-us/.
18.Інтернет Ресурс Microsoft,URL: https://docs.microsoft.com/.
19.Інтернет Ресурс GTKWave,URL: http://gtkwave.sourceforge.net/.
20.Інтернет Ресурс Icarus,URL: http://bleyer.org/icarus/.

ДОДАТКИ

ДОДАТОК 1

Вихідний код транслятора

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CompilerApp
{
    abstract class INode
    {
        List<INode> childs;

        public INode()
        {
            childs = new List<INode>();
        }

        public bool AddNode(INode node)
        {
            if (null != node)
            {
                childs.Add(node);
                return true;
            }
            return false;
        }

        abstract public bool Empty();

        public bool ChildEmpty()
        {
            return (childs.First<INode>().Empty());
        }

        abstract public void Write(string tabs);

        public List<INode> getChilds()
        {
            return childs;
```

```
    }

    abstract public string getGrammar();
    abstract public string getIdentifier();
    abstract public Lexem getLexem();
}

class LexemNode:INode
{
    Lexem lexem;

    public LexemNode(Lexem lexem)
    {
        this.lexem = lexem;
    }

    public override bool Empty()
    {
        return false;
    }

    public override void Write(string tabs)
    {
        Console.WriteLine(tabs + lexem.ToString());
    }

    public override string getGrammar()
    {
        return null;
    }

    public override string getIdentifier()
    {
        return lexem.identifier;
    }

    public override Lexem getLexem()
    {
        return lexem;
```

```
    }
}

class GrammarNode:INode
{
    string grammar;

    public GrammarNode(string grammar)
    {
        this.grammar = grammar;
    }

    public override bool Empty()
    {
        return (grammar == "<empty>");
    }

    public override void Write(string tabs)
    {
        Console.WriteLine(tabs + grammar);
    }

    public override string getGrammar()
    {
        return grammar;
    }

    public override string getIdentifier()
    {
        return null;
    }

    public override Lexem getLexem()
    {
        return default(Lexem);
    }
}

class Syntaxer
```

```
        {
            private static void WriteSyntaxerResult(INode node, string tabs = "")
            {
                node.Write(tabs);
                foreach(INode child in node.getChilds())
                {
                    WriteSyntaxerResult(child, tabs + "    ");
                }
            }

            private static INode Check(List<Lexem> lexems, string token, bool printerr =
true)
            {
                Lexem lexem = lexems.FirstOrDefault<Lexem>();
                if (null == lexem.identifier)
                {
                    if (printerr)
                    {
                        Console.WriteLine("Unexpected end of file");
                    }
                    return null;
                }
                if (token == lexem.identifier)
                {
                    lexems.RemoveAt(0);
                    return new LexemNode(lexem);
                }
                if (printerr)
                {
                    Console.WriteLine("Unexpected token " + lexem.identifier + ", '" +
token + "' is missed: line " + lexem.line + "; col " + lexem.col);
                }
                return null;
            }

            private static INode CheckCommand(List<Lexem> lexems, Dictionary<string, int>
identifiers, string type, bool printerr = true)
            {
                Lexem lexem = lexems.FirstOrDefault<Lexem>();
```

```
                if (null == lexem.identifier)
                {
                    if (printerr)
                    {
                        Console.WriteLine("Unexpected end of file");
                    }
                    return null;
                }
                if (404 < lexem.code && lexem.code < 500)
                {
                    GrammarNode grammarP, grammarC;
                    LexemNode lexemNode = new LexemNode(lexem);
                    grammarP = new GrammarNode(type);
                    grammarC = new GrammarNode("<identifier>");
                    grammarC.AddNode(lexemNode);
                    grammarP.AddNode(grammarC);

                    lexems.RemoveAt(0);
                    return grammarP;
                }
                if (printerr)
                {
                    Console.WriteLine("Command expected, found " + lexem.identifier + ":
line " + lexem.line + "; col " + lexem.col);
                }
                return null;
            }

            private static INode CheckIdentifier(List<Lexem> lexems, Dictionary<string,
int> identifiers, string type, bool printerr = true)
            {
                Lexem lexem = lexems.FirstOrDefault<Lexem>();
                if (null == lexem.identifier)
                {
                    if (printerr)
                    {
                        Console.WriteLine("Unexpected end of file");
                    }
                    return null;
```

```
                }
                if (identifiers.Keys.Contains<string>(lexem.identifier))
                {
                    GrammarNode grammarP, grammarC;
                    LexemNode lexemNode = new LexemNode(lexem);
                    grammarP = new GrammarNode(type);
                    grammarC = new GrammarNode("<identifier>");
                    grammarC.AddNode(lexemNode);
                    grammarP.AddNode(grammarC);

                    lexems.RemoveAt(0);
                    return grammarP;
                }
                if (printerr)
                {
                    Console.WriteLine("Identifier expected, found " + lexem.identifier +
": line " + lexem.line + "; col " + lexem.col);
                }
                return null;
            }

            private static INode CheckConstant(List<Lexem> lexems, Dictionary<string,
int> constants, bool printerr = true)
            {
                Lexem lexem = lexems.FirstOrDefault<Lexem>();
                if (null == lexem.identifier)
                {
                    if (printerr)
                    {
                        Console.WriteLine("Unexpected end of file");
                    }
                    return null;
                }
                if (constants.Keys.Contains<string>(lexem.identifier))
                {
                    GrammarNode grammar = new GrammarNode("<unsigned-integer>");
                    LexemNode lexemNode = new LexemNode(lexem);
                    grammar.AddNode(lexemNode);
```

```
                    lexems.RemoveAt(0);
                    return grammar;
                }
                if (printerr)
                {
                    Console.WriteLine("Identifier expected, found " + lexem.identifier +
": line " + lexem.line + "; col " + lexem.col);
                }
                return null;
            }

            private static INode Declaration(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants, bool canEmpty = false)
            {
                GrammarNode node = new GrammarNode("<declaration>");
                if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<variable-
identifier>", !canEmpty)))
                {
                    if (canEmpty)
                        return new GrammarNode("<empty>");
                    return null;
                }
                if (!node.AddNode(Check(lexems, "=")))
                {
                    return null;
                }
                if (!node.AddNode(CheckConstant(lexems, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "\n")))
                {
                    return null;
                }
                return node;
            }

            private static INode DeclarationsList(List<Lexem> lexems, Dictionary<string,
int> identifiers, Dictionary<string, int> constants)
```

```
        {
            GrammarNode node = new GrammarNode("<declarations-list>");
            if (!node.AddNode(Declaration(lexems, identifiers, constants, true)))
            {
                return null;
            }
            if (node.ChildEmpty())
            {
                return node;
            }
            if (!node.AddNode(DeclarationsList(lexems, identifiers, constants)))
            {
                return null;
            }
            return node;
        }

        private static INode Declarations(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants)
        {
            GrammarNode grammarP = new GrammarNode("<declarations>");
            GrammarNode grammarC = new GrammarNode("<variable-declarations>");
            if (!grammarC.AddNode(Check(lexems, "VARS", false)))
            {
                GrammarNode grammar = new GrammarNode("<empty>");
                grammarC.AddNode(grammar);
                grammarP.AddNode(grammarC);
                return grammarP;
            }
            if (!grammarC.AddNode(Check(lexems, "\n")))
            {
                return null;
            }
            if (!grammarC.AddNode(Declaration(lexems, identifiers, constants)))
            {
                return null;
            }
            if (!grammarC.AddNode(DeclarationsList(lexems, identifiers, constants)))
            {
```

```
                    return null;
                }
                grammarP.AddNode(grammarC);
                return grammarP;
            }

            private static INode Statement(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants, bool canEmpty = false)
            {
                GrammarNode node = new GrammarNode("<statement>");
                if (!node.AddNode(CheckCommand(lexems, identifiers, "<operation-
identifier>", !canEmpty)))
                {
                    if (canEmpty)
                        return new GrammarNode("<empty>");
                    return null;
                }
                if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<variable-
identifier>", false)) &&
                    !node.AddNode(CheckConstant(lexems, constants, false)) &&
!node.AddNode(Check(lexems, "_")))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, ",")))
                {
                    return null;
                }
                if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<variable-
identifier>", false)) &&
                    !node.AddNode(CheckConstant(lexems, constants, false)) &&
!node.AddNode(Check(lexems, "_")))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, ",")))
                {
                    return null;
                }
```

```
                if (!node.AddNode(CheckConstant(lexems, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "[")))
                {
                    return null;
                }
                if (!node.AddNode(CheckConstant(lexems, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "]")))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "\n")))
                {
                    return null;
                }
                return node;
            }

            private static INode StatementsList(List<Lexem> lexems, Dictionary<string,
int> identifiers, Dictionary<string, int> constants)
            {
                GrammarNode node = new GrammarNode("<statements-list>");
                if (!node.AddNode(Statement(lexems, identifiers, constants, true)))
                {
                    return null;
                }
                if (node.ChildEmpty())
                {
                    return node;
                }
                if (!node.AddNode(StatementsList(lexems, identifiers, constants)))
                {
                    return null;
                }
```

```
                return node;
            }

            private static INode Block(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants)
            {
                GrammarNode node = new GrammarNode("<block>");
                if (!node.AddNode(Declarations(lexems, identifiers, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "BEGIN")))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "\n")))
                {
                    return null;
                }
                if (!node.AddNode(StatementsList(lexems, identifiers, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "END")))
                {
                    return null;
                }
                return node;
            }

            private static INode Program(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants)
            {
                GrammarNode node = new GrammarNode("<program>");
                if (!node.AddNode(Check(lexems, "PROGRAM")))
                {
                    return null;
                }
```

```
                if (!node.AddNode(CheckIdentifier(lexems, identifiers, "<procedure-
identifier>")))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "\n")))
                {
                    return null;
                }
                if (!node.AddNode(Block(lexems, identifiers, constants)))
                {
                    return null;
                }
                if (!node.AddNode(Check(lexems, "\n")))
                {
                    return null;
                }
                return node;
            }

            public static INode MakeTree(List<Lexem> lexems, Dictionary<string, int>
identifiers, Dictionary<string, int> constants)
            {
                GrammarNode tree = new GrammarNode("<vlm-program>");
                if (tree.AddNode(Program(lexems, identifiers, constants)))
                {
                    if (lexems.Count == 0)
                    {
                        WriteSyntaxerResult(tree);
                        return tree;
                    }
                    Console.WriteLine("Unexpected code in the end of file");
                }
                return null;
            }
        }
    }
```

ДОДАТОК 2

Вихідний код емуляційної частини

```
using Syncfusion.UI.Xaml.Diagram;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using EmulatorB.Elements;
using EmulatorB.AscoElements;
using System.ComponentModel;

namespace EmulatorB
{


    public partial class Canva : UserControl, INotifyPropertyChanged
    {

        private string STEP_;
        public string STEP
        {
            get { return STEP_; }
            set { STEP_ = value; OnPropertyChanged("STEP"); }
        }

        int last;
        List<ProcessorVisualElement> Processors = new List<ProcessorVisualElement>();
        List<InputVisualElement> Inputs = new List<InputVisualElement>();
        List<OutputVisualElement> Outputs = new List<OutputVisualElement>();
        AsocMemoryVisualElement AsocMemory = new AsocMemoryVisualElement();
        BKVisualElement MainBK = new BKVisualElement();
```

```
BKVisualElement SaveBK = new BKVisualElement();
List<RegisterVisualElement> Registers = new List<RegisterVisualElement>();
ControlBlock Control;
ComutatorVisualElement Comutator = new ComutatorVisualElement();
List<LineCSV> lines;

int processors;
int inputs = 3;
int outputs = 3;
public Canva(int counter, List<LineCSV> lines)
{
    this.lines = lines;
    last = lines.Count;
    processors = counter;
    InitializeComponent();
    Control = new ControlBlock(processors);
    InitProcessors();
    InitInputs();
    InitAsocMemory();
    InitMainBK();
    InitSaveBK();
    InitOutputs();
    InitRegisters();
    InitControlBlock();
    InitConnectionInputsAsocMemory();
    InitConnectionsAsocMmeorySaveBK();
    InitConnectionsAsocMmeoryMainBK();
    InitConnectionsMainBKOutputs();
    InitConnectionsSaveBKRegisters();
    InitConnectionsMainBKRegisters();
    InitConnectionsRegistersControlBlock();
    InitConnectionsRegistersProcessors();
    InitComutator();
    InitConnectionsProcComutator();
    InitConnectionsComutatorAsoc();
    DataContext = this;
}
```

```
        public void InitProcessors()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel Proc = new NodeViewModel();
                Proc.OffsetX = 1000;
                Proc.OffsetY = 50 + 100 * i;
                Proc.ID = "proc" + i;

                ProcessorVisualElement PVE = new ProcessorVisualElement("Processor "
+ i);
                Proc.Content = PVE;
                NodeCollection.Add(Proc);
                Processors.Add(PVE);
            }
        }

        public void InitRegisters()
        {
            for (int i = 0; i < processors; i++)
            {
                NodeViewModel Node = new NodeViewModel();
                Node.OffsetX = 800;
                Node.OffsetY = 50 + 100 * i;
                Node.ID = "reg" + i;

                RegisterVisualElement PVE = new RegisterVisualElement();
                Node.Content = PVE;
                NodeCollection.Add(Node);
                Registers.Add(PVE);
            }
        }



        public void InitAsocMemory()
        {
            NodeViewModel Node = new NodeViewModel();
            Node.OffsetX = 200;
```

```
    Node.OffsetY = 200;
    Node.ID = "asoc";

    Node.Content = AsocMemory;
    NodeCollection.Add(Node);
}
public void InitControlBlock()
{
    NodeViewModel Node = new NodeViewModel();
    Node.OffsetX = 800;
    Node.OffsetY = 600;
    Node.ID = "control";

    Node.Content = Control;
    NodeCollection.Add(Node);
}



public void InitMainBK()
{
    NodeViewModel Node = new NodeViewModel();
    Node.OffsetX = 200;
    Node.OffsetY = 400;
    Node.ID = "MainBK";

    Node.Content = MainBK;
    NodeCollection.Add(Node);
}

public void InitSaveBK()
{
    NodeViewModel Node = new NodeViewModel();
    Node.OffsetX = 550;
    Node.OffsetY = 400;
    Node.ID = "SaveBK";

    Node.Content = SaveBK;
    NodeCollection.Add(Node);
```

```
}


public void InitComutator()
{
    NodeViewModel Node = new NodeViewModel();
    Node.OffsetX = 650;
    Node.OffsetY = 20;
    Node.ID = "com";

    Node.Content = Comutator;
    NodeCollection.Add(Node);
}



public void InitInputs()
{
    for (int i = 0; i < inputs; i++)
    {
        NodeViewModel Input = new NodeViewModel();
        Input.OffsetX = 180 + 170 * i;
        Input.OffsetY = 10;
        Input.ID = "input" + i;
        InputVisualElement IVE = new InputVisualElement("Input " + i);
        Inputs.Add(IVE);
        Input.Content = IVE;
        NodeCollection.Add(Input);
    }
}


public void InitOutputs()
{
    for (int i = 0; i < outputs; i++)
    {
        NodeViewModel output = new NodeViewModel();
        output.OffsetX = 100 + i * 170;
        output.OffsetY = 600;
```

```
                output.ID = "output" + i;
                OutputVisualElement IVE = new OutputVisualElement("Output " + i);
                Outputs.Add(IVE);
                output.Content = IVE;
                NodeCollection.Add(output);
            }
        }

        void InitConnectionInputsAsocMemory()
        {
            for (int i = 0; i < inputs; i++)
            {
                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "input" + i;
                CVM.TargetNodeID = "asoc";

                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);
            }
        }

        void InitConnectionsAsocMmeorySaveBK()
        {
            ConnectorViewModel CVM = new ConnectorViewModel();
            CVM.SourceNodeID = "asoc";
            CVM.TargetNodeID = "SaveBK";

            CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
            ConectorCollection.Add(CVM);
        }


        void InitConnectionsAsocMmeoryMainBK()
        {
            ConnectorViewModel CVM = new ConnectorViewModel();
            CVM.SourceNodeID = "asoc";
            CVM.TargetNodeID = "MainBK";
```

```
                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);

            }

            void InitConnectionsMainBKOutputs()
            {
                for (int i = 0; i < outputs; i++)
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "MainBK";
                    CVM.TargetNodeID = "output" + i;

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);
                }
            }


            void InitConnectionsProcComutator()
            {

                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "proc0";
                CVM.TargetNodeID = "com";

                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);
            }

            void InitConnectionsComutatorAsoc()
            {

                ConnectorViewModel CVM = new ConnectorViewModel();
                CVM.SourceNodeID = "com";
```

```
                CVM.TargetNodeID = "asoc";

                CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                ConectorCollection.Add(CVM);
            }

            void InitConnectionsMainBKRegisters()
            {
                for (int i = 0; i < processors; i++)
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "MainBK";
                    CVM.TargetNodeID = "reg" + i;

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);
                }
            }


            void InitConnectionsSaveBKRegisters()
            {
                for (int i = 0; i < processors; i++)
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "SaveBK";
                    CVM.TargetNodeID = "reg" + i;

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);
                }
            }

            void InitConnectionsRegistersControlBlock()
            {
                for (int i = 0; i < processors; i++)
```

```
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "reg" + i;
                    CVM.TargetNodeID = "control";

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);
                }
            }

            void InitConnectionsRegistersProcessors()
            {
                for (int i = 0; i < processors; i++)
                {
                    ConnectorViewModel CVM = new ConnectorViewModel();
                    CVM.SourceNodeID = "reg" + i;
                    CVM.TargetNodeID = "proc" + i;

                    CVM.ConnectorGeometryStyle = this.FindResource("ConnectorStyle") as
Style;
                    ConectorCollection.Add(CVM);
                }
            }

            int takt = 0;
            int globaltakt = 0;
            int state = 0;
            LineCSV line;
            int lastline = 100;
            private void Button_Click(object sender, RoutedEventArgs e)
            {
                MachineTackt();
            }

            private void MachineTackt()
            {
                if (Outputs[0].rdy)
                {
```

```
    MessageBox.Show("END");
    return;
}
if (takt == last)
{
    foreach (var c in Processors)
    {
        c.Progress.Value = 0;
        c.Clear();

    }
    SaveBK.Clear();
    SaveBK.Clear();
    WriteOutPut();
    globaltakt++;
    STEP = globaltakt.ToString();
    return;
}

line = lines[takt];

checkProcessors();
Control.Decrement();

if (!Fool())
{
    globaltakt++;
    return;
}
switch (state)
{
    case 0:
        {
            if (takt >= 6)
            {
                clearInputs();
            }
            else
                checkInputs();
```

```
            state = 1;
            break;
        }
    case 1:
        {
            SetAsoc();
            state = 2;
            break;
        }
    case 2:
        {
            if (asoccome % 3 != 0)
            {
                state = 1;

                if (takt >= 6)
                {
                    clearInputs();
                }
                else
                    checkInputs();
            }
            else
            {
                SetCmd();
                state = 3;
            }
            break;
        }
    case 3:
        {
            if (lastline != takt)
            {
                UseProc();
                lastline = takt;
            }
            state = 4;
            break;
        }
```

```
            case 4:
                {
                    takt++;
                    state = 0;
                    break;
                }
            default:
                {
                    break;
                }
        }

        globaltakt++;
        STEP = globaltakt.ToString();
    }

    public void WriteOutPut()
    {
        Outputs[0].MoveProgress(lines.Last().operand1.ToString());
    }

    void checkTimers()
    {
        for (int i = 0; i < processors; i++)
        {
            Registers[i].Decrement();
        }
    }

    private bool Fool()
    {
        foreach (var c in Processors)
        {
            if (c.isfree)
                return true;
        }
        return false;
    }
```

```
        void checkProcessors()
        {
            foreach (var c in Processors)
            {
                if (!c.isfree && c.isWork)
                {
                    c.MoveProgress();
                }
            }
            //  return false;
        }

        int curreProc = 0;
        void UseProc()
        {
            if (curreProc >= Processors.Count)
            {
                curreProc = 0;
                return;
            }
            if (!Processors[curreProc].isWork)
            {
                curreProc++;
                return;
            }
            Registers[curreProc].SetValue(lasttime.ToString(), line.name);
            Control.SetTime(curreProc, lasttime.ToString());
            Processors[curreProc].SetProc("0", line.num.ToString(), "0",
line.operand1.ToString(), "0");
            Processors[curreProc].MoveProgress();

            curreProc++;
        }

        void checkInputs()
        {

            Inputs[0].WRITE("1", line.num.ToString(), "0", line.name.ToString(),
line.next.ToString(), "0");
```

```
                    Inputs[1].WRITE("1", line.num.ToString(), "0", line.operand1.ToString(),
line.next.ToString(), "0");
                    Inputs[2].WRITE("1", line.num.ToString(), "0", line.operand2.ToString(),
line.next.ToString(), "0");
                }

                private void clearInputs()
                {
                    Inputs[0].WRITE("", "", "0", "", "", "0");
                    Inputs[1].WRITE("", "", "0", "", "", "0");
                    Inputs[2].WRITE("", "", "0", "", "", "0");
                }

                int asoccome = 0;
                int lasttime;
                private void SetAsoc()
                {
                    asoccome++;
                    Random rand = new Random();
                    int a = rand.Next(10, 20);
                    lasttime = a;
                    if (a < 15)
                    {
                        AsocMemory.AddStroke1("1", line.name, "0");
                        AsocMemory.AddStroke2("0", "0", "0");
                        AsocMemory.AddStroke3("1", "0", "0", "0", "0", "0");
                    }
                    else
                    {
                        AsocMemory.AddStroke1("1", "0", "0");

                        AsocMemory.AddStroke2("0", line.operand1.ToString(), "0");
                        AsocMemory.AddStroke3("1", line.operand2.ToString(), "0",
line.next.ToString(), line.num.ToString(), lasttime.ToString());
                    }
                }

                int k = 0;
                int kostil = 0;
```

```
        private void SetCmd()
        {
            MainBK.Set(line.name, line.operand1.ToString(), line.operand2.ToString(),
line.next.ToString());
            if (!CheckPorcessors() && kostil == 0)
            {
                SaveBK.Set(line.name, line.operand1.ToString(),
line.operand2.ToString(), line.next.ToString());
                kostil = 1;
            }
            else
            {
                SaveBK.Set("", "", "", "");

            }
            /*
            if( k % 2 != 0)
            {
                SaveBK.Set(line.name, line.operand1.ToString(),
line.operand2.ToString(), line.next.ToString());
            }*/
        }

        bool CheckPorcessors()
        {
            foreach(var c in Processors)
            {
                if (!c.isWork)
                {
                    k++;
                    return false;
                }
            }
            return true;
        }

        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged(string prop = "")
        {
```

```
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}
```

ДОДАТОК 3

Копія презентації

ПРОГРАМНА МОДЕЛЬ ВІДМОВОСТІЙКОЇ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ, ЩО КЕРУЄТЬСЯ ПОТОКОМ ДАНИХ

Виконав: Вінник Денис Андрійович

Науковий керівник: к.т.н. доц. Жабіна Валентина Валеріївна

Київ – 2018

АКТУАЛЬНІСТЬ

- ПРОБЛЕМА ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ СИСТЕМ НЕДОСТАТНЬО ДОСЛІДЖЕНА ДЛЯ СПД ЧЕРЕЗ СПЕЦИФІКУ ОРГАНІЗАЦІЇ В НИХ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ.

ПРОБЛЕМАТИКА

- Для систем реального часу часові витрати при програмних засобах підвищення відмовостійкості є неприйнятними.
- В даному випадку більш прийнятними є методи динамічної реконфігурації систем при відмові обладнання.

ОБ'ЄКТ І ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Об'єктом дослідження** є процес обробки даних у відмовостійких обчислювальних системах, що керуються потоком даних.
- **Предметом дослідження** є методи забезпечення відмовостійкості в обчислювальних системах, що керуються потоком даних.

МЕТА ДОСЛІДЖЕННЯ

- **МЕТОЮ** ДОСЛІДЖЕННЯ Є РЕАЛІЗАЦІЯ ПРОГРАМНОЇ МОДЕЛІ ДЛЯ ТЕСТУВАННЯ ВІДМОВОСТІЙКОЇ СИСТЕМИ ДЛЯ ВИЗНАЧЕННЯ ЇЇ ОПТИМАЛЬНИХ ПАРАМЕТРІВ.

ЗАДАЧІ

- – АНАЛІЗ ТА ДОСЛІДЖЕННЯ ІСНУЮЧИХ МЕТОДІВ І ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ СИСТЕМ З МЕТОЮ ВИЯВЛЕННЯ ЇХ НЕДОЛІКІВ І ВИБОРУ НАЙБІЛЬШ ДОЦІЛЬНОГО ВАРІАНТУ ДЛЯ СПД;
- – АНАЛІЗ МЕТОДІВ ДИНАМІЧНОЇ РЕКОНФІГУРАЦІЇ СИСТЕМ;
- – РОЗРОБКА АРХІТЕКТУРИ ВІДМОВОСТІЙКОЇ СПД З ВИКОРИСТАННЯМ АСОЦІАТИВНОЇ ПАМ'ЯТІ;
- – РОЗРОБКА ПРОГРАМНОЇ МОДЕЛІ ЕМУЛЯТОРА РОБОТИ АРХІТЕКТУРИ;
- – ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОЇ МОДЕЛІ ЗА ДОПОМОГОЮ ЕМУЛЯТОРА.

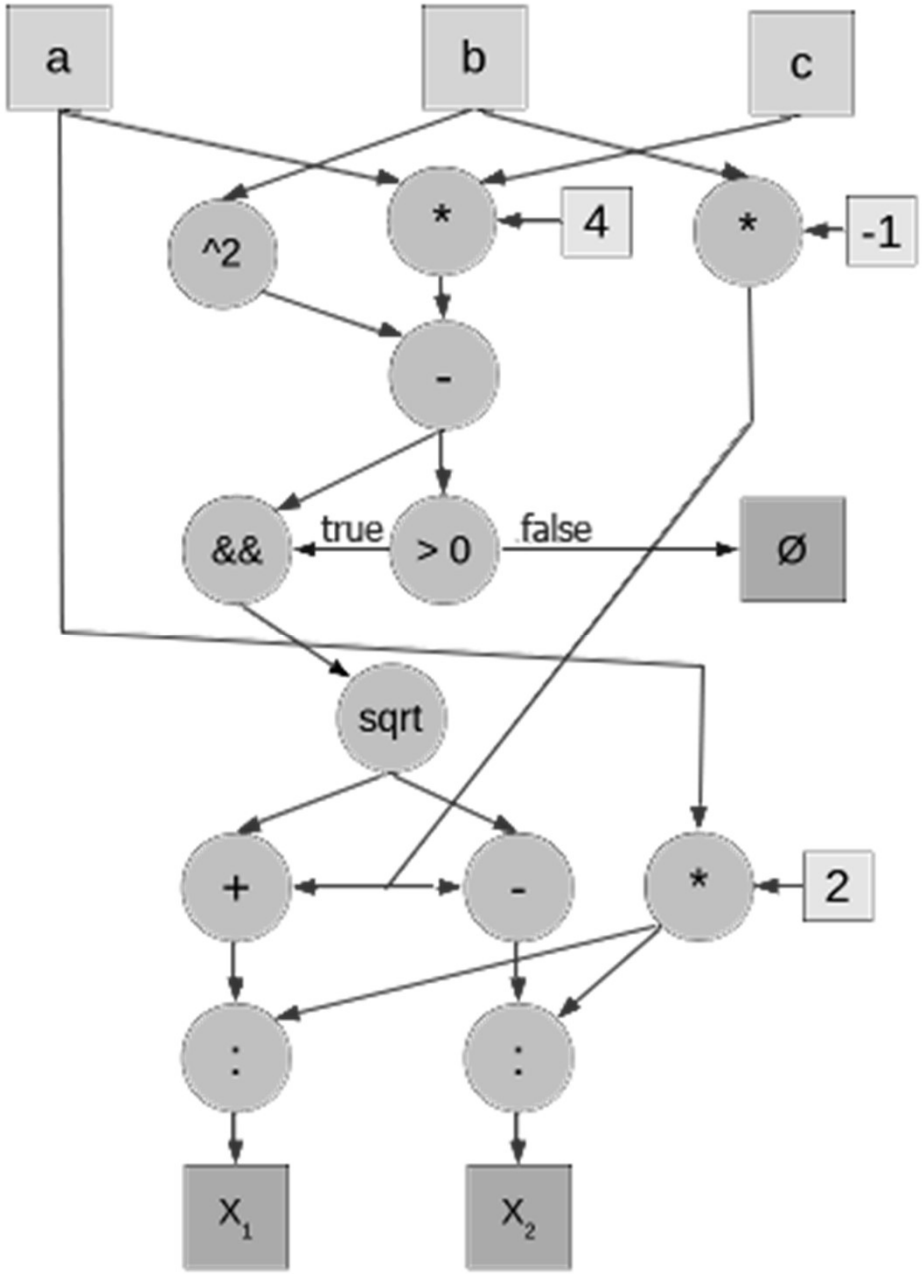
СПД

- **СПД** – ЦЕ СИСТЕМИ, ЩО ВИКОРИСТОВУЮТЬ МЕХАНІЗМ УПРАВЛІННЯ ОБЧИСЛЕННЯМИ, ПРИ ЯКОМУ КОМАНДИ ВИКОНУЮТЬСЯ ТОДІ, КОЛИ СТАЮТЬ ДОСТУПНИМИ ЇХ ОПЕРАНДИ.

ФОРМАТ АКТОРА

Тип слова	Номер операції	Номер операнда	КОП	Номер наступної операції	Признак константи	Порядок введення

ГРАФ ПОТОКУ ДАНИХ



ВІДМОВОСТІЙКІСТЬ

- **Відмовостійкість** – це властивість архітектури ОС, що забезпечує виконання заданих функцій у випадках, коли в апаратних і програмних засобах системи виникають відмови.

ЗАБЕЗПЕЧЕННЯ ВІДМОВОСТІЙКОСТІ

- В даній роботі розглядається наступний підхід до забезпечення відмовостійкості СПД. Для визначення несправностей ОМ використовується контроль часових інтервалів, що визначені для кожної операції. Контролюючим об'єктом є апаратно захищене ядро системи.

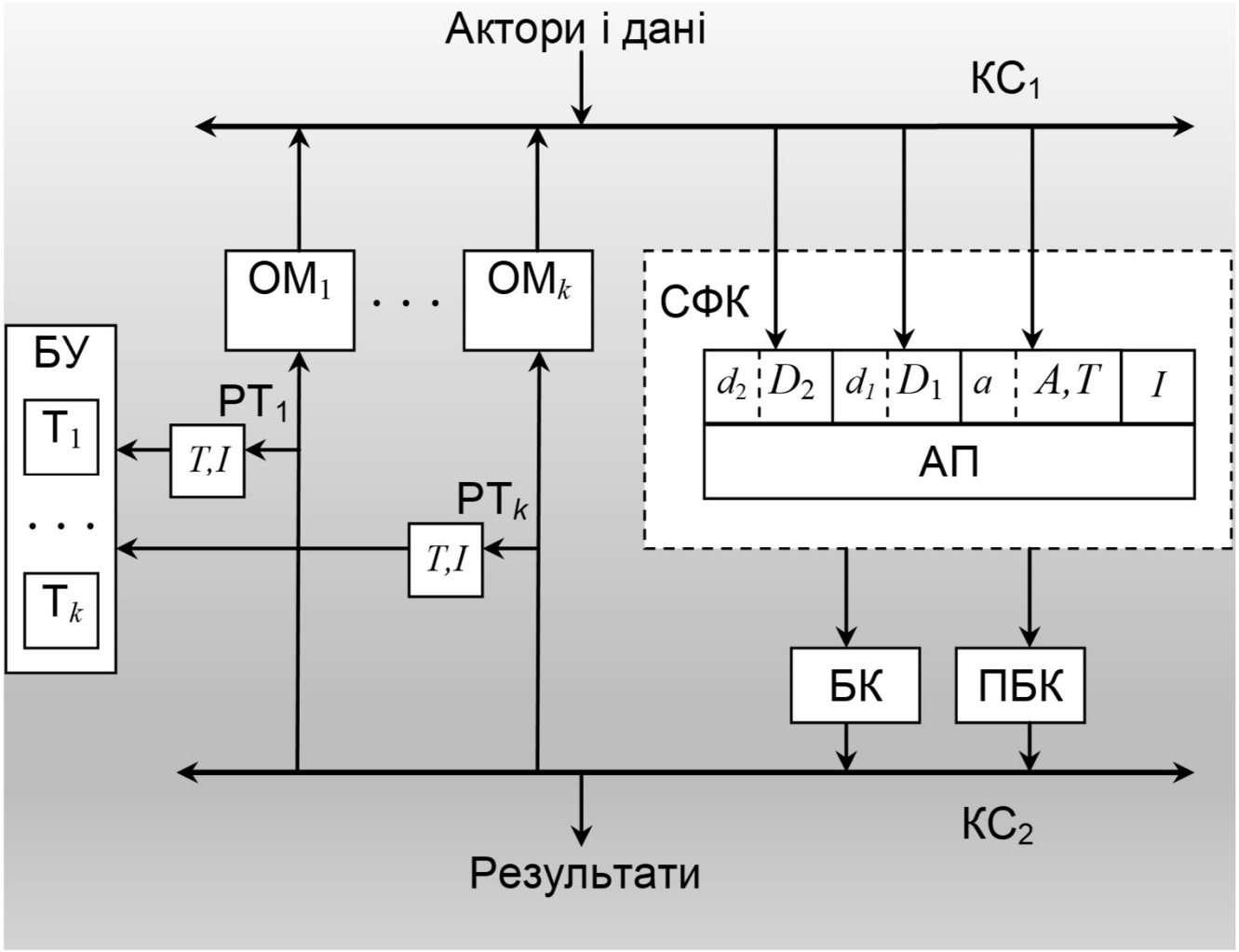
СКЛАД СИСТЕМИ

- Обчислювальні модулі (ОМ)
- Середовище формування команд (СФК)
- Асоціативна пам’ять (АП)
- Регістри, для тимчасового зберігання інформації (РТ)
- Блок управління (БУ)
- Буферна пам’ять команд (БК)
- Комутаційні середовища (КС)

АСОЦІАТИВНА ПАМ’ЯТЬ

- **Асоціативна пам’ять** – особливий вид машинної пам'яті, відома також як пам'ять, що адресується за вмістом.

АРХІТЕКТУРА ВІДМОВОСТІЙКОЇ СПД

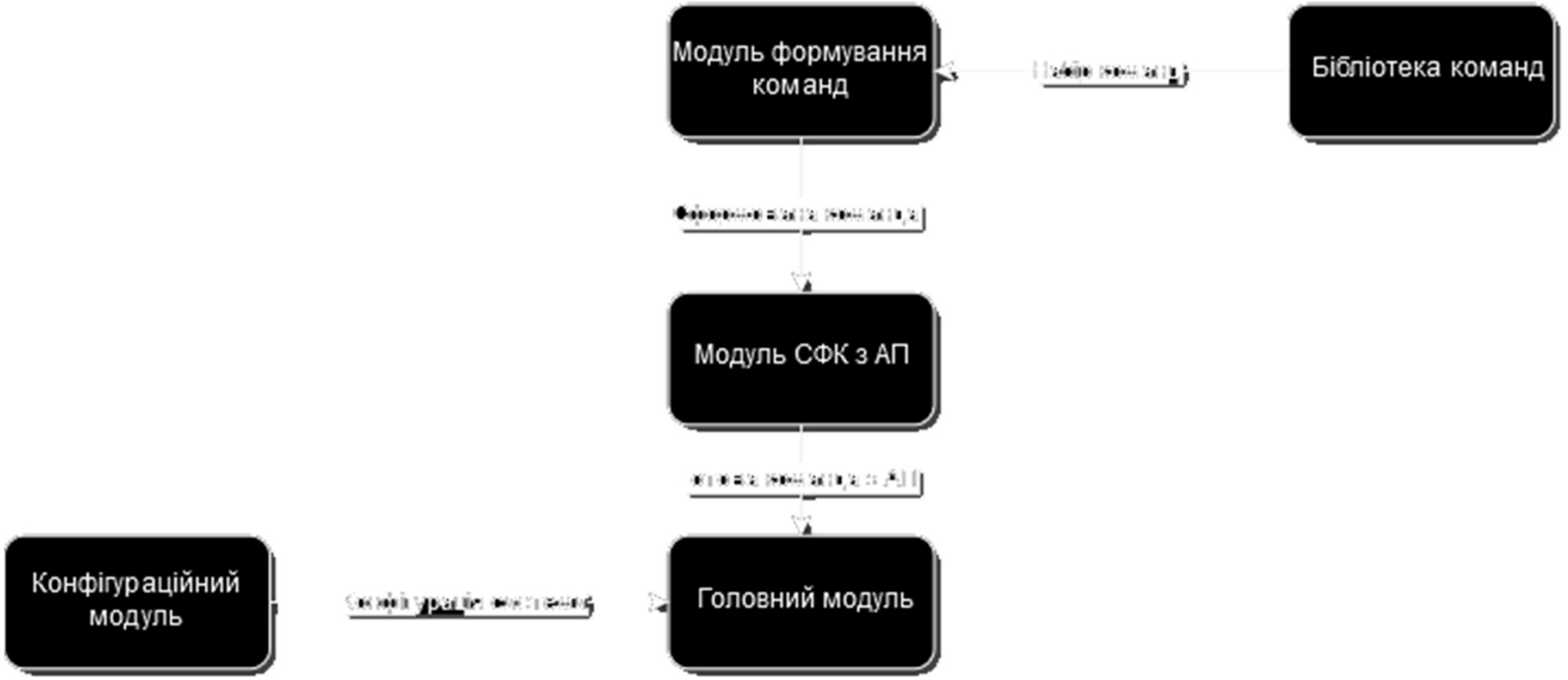


ІСНУЮЧІ АНАЛОГИ

- Існуючі аналоги програмних моделей, такі як Xilinx, Altera і тд, є надто громіздкими, дорогими і незрозумілими для новачків.

ІНСТРУМЕНТИ

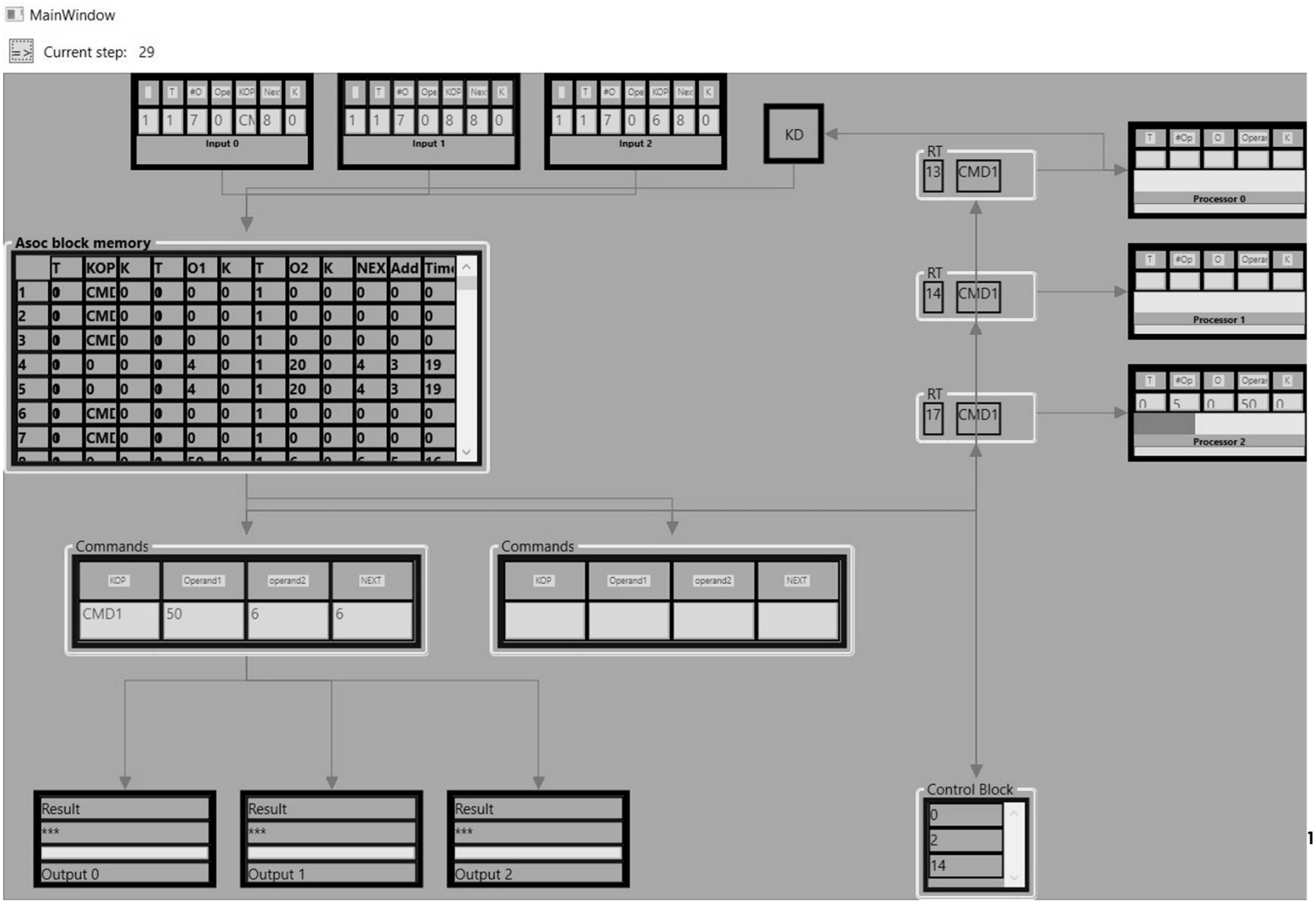
ПРОГРАМНА МОДЕЛЬ



БІБЛІОТЕКА КОМАНД

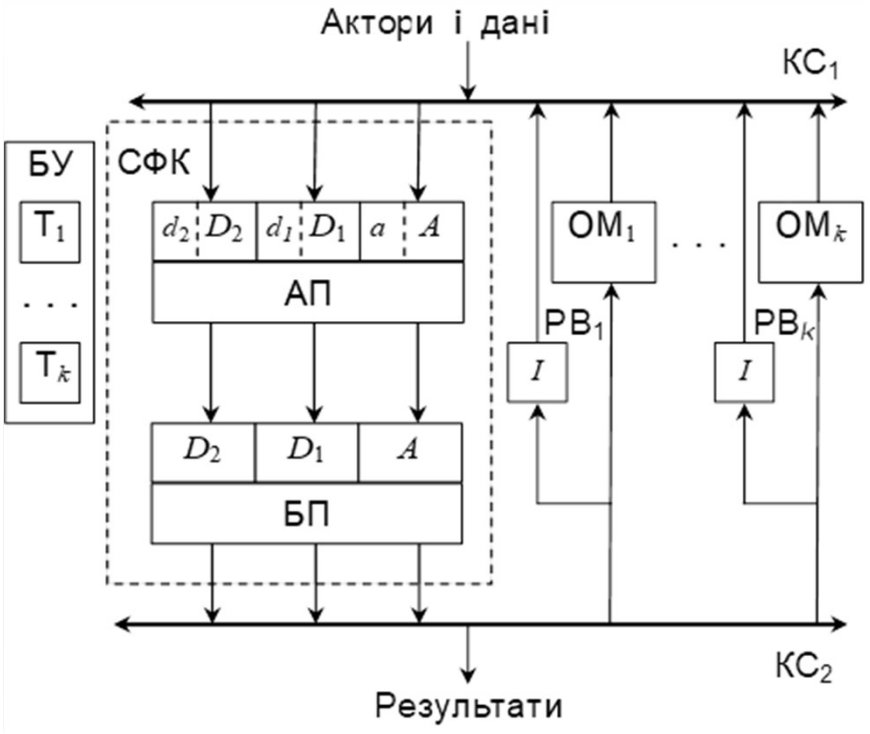
- PROGRAM TEST
- VARS
- A = 10
- BEGIN
- CMD1 OBJ1,20, 2[0]
- OUT _, 1, 2[0]
- END

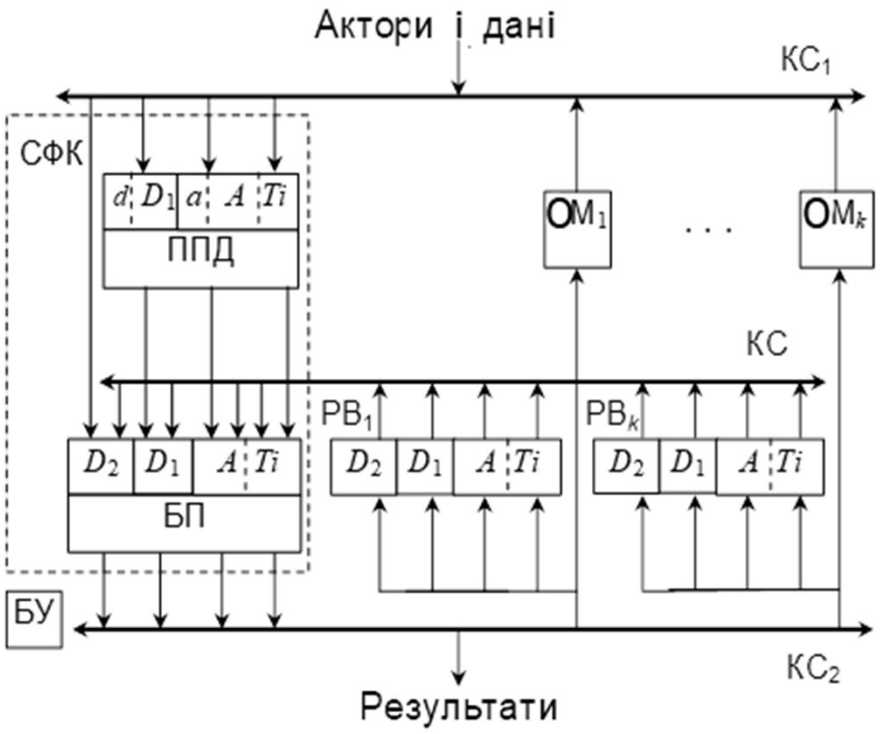
ПРОГРАМНА МОДЕЛЬ



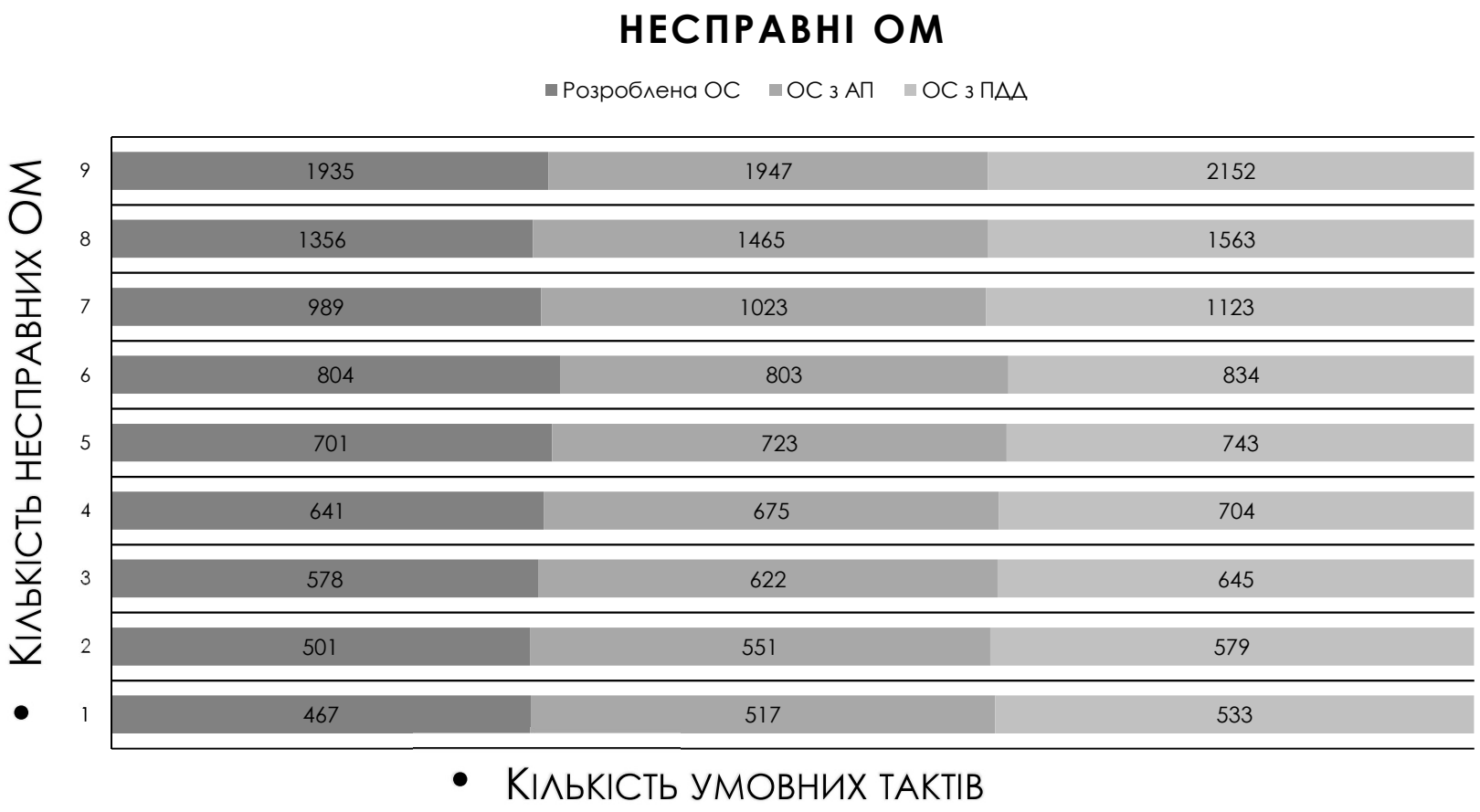
IDEA

АРХІТЕКТУРИ ДЛЯ ПОРІВНЯННЯ





АНАЛІЗ РЕЗУЛЬТАТІВ



НАДІЙНІСТЬ

- Інтенсивність відмов величина адитивна і визначається для кожного компонента як:
- $\lambda = \Sigma \lambda_I * N_I$,
- де λ_I - інтенсивність відмов модулів, роз'ємом і пайок,
- N_I - кількість відповідних компонент.
- $K_Г = 1 / (1 + \lambda * T_{ВОССТ}) = 0{,}99973878$

НАУКОВА НОВИЗНА

- Запропонована модифікація методу забезпечення відмовостійкості потокових обчислювальних систем, яка відрізняється від існуючих методів врахуванням реальної швидкості виконання кожної операції, що дозволяє зменшити час на виявлення несправного обчислювального модуля. В системах-аналогах враховується лише максимальний час виконання операції.
- Реалізована гнучка програмна модель розробленої архітектури обчислювальної системи із засобами відмовостійкості.
- Запропонована система команд для відмово стійкої системи, яка може розширюватися за допомогою спеціального конфігураційного файлу

БІЗНЕС-МОДЕЛЬ

Бізнес-модель

Ключові партнери

Компанії, що займаються наукою про дані.
НТУУ "КПІ" ім. І. Сікорського

Ключова діяльність

Проектування продукту
Розробка продукту
Подальша підтримка продукту

Ключові ресурси

Розробник
Фінанси
Розроблений метод
Платформи для тестування

Ціннісна пропозиція

Підвищення ефективності реалізації паралельних обчислень у розподілених системах.
Зменшення витрат апаратного забезпечення.
Зменшення затримок при обробці даних.

Взаємовідносини з клієнтами

Допомога та/або консультація при встановленні продукту.
Подальша підтримка зв'язку для покращення продукту.
Надання подальших оновлень

Канали розповсюдження

Різного роду репозиторії для бібліотек з відкритим кодом.

Сегмент користувачів

Компанії з великим обігом інформації.
Компанії, що використовують розподілені системи.
Компанії, що мають справу з наукою про дані.

Структура витрат

Оплата сервісів і патентування
Витрати на розробку та необхідні ресурси
Витрати на рекламу та інші методи просування
Витрати на подальшу підтримку

Потоки доходів

Оплата за продукт
Оплата за встановлення та налаштування продукту
Оплата подальшої підтримки та оновлень

ВИСНОВКИ

- Проаналізовано та досліджено існуючі методи і засоби забезпечення відмовостійкості систем, виявлено їх недоліки та вибрано метод часових інтервалів, як найбільш доцільний варіант для СПД;
- Проаналізовано методи динамічної реконфігурації систем;
- Розроблено архітектуру відмовостійкої СПД з використанням асоціативної пам'яті;
- Розроблено програмну модель для емуляції роботи архітектури;
- Експериментально досліджено ефективність запропонованої архітектури за допомогою емулятора. Порівняння показало, що розроблена архітектура на обраному алгоритмі має пришвидшення до 15%;
- Розрахований коефіцієнт готовності 0.9997 є цілком достатнім показником надійності.

ДЯКУЮ ЗА УВАГУ